JAX-RS 39 | JAVA INTERFACES 47 | FANTOM 53

Java™ magazine

By and for the Java community

SEPTEMBER/OCTOBER 2016

Devices and IoT

14 INTERACTING WITH SENSORS

24 CONTROLLING CNC ROUTING FROM A RASPBERRY PI

32 IOT AND THE CLOUD

ORACLE®

//table of contents /

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Account Manager
Marcin Gamza

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.


XRebel

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

Appreciating Limited Choice in Languages

The more prescriptive a language is in the details, the easier it is to code productively.

Our coverage of JDK Enhancement Proposals (JEPs) in this issue examines a recent proposal to standardize the syntax of command-line arguments for tools that ship in the JDK. As the proposal points out in support of its core concern, presently there are multiple ways of asking for help from the command line. And if you happen to guess wrong when using a given tool, you need to circle through the variety of possibilities. These can vary from `-help` to `--help` to `-?`. And then there's the unmentioned last resort, which is to run the program with no arguments and see what kind of information you get in the error message.

I wholly support this standardization, but I'd go much further. In my view, the syntax of command-line switches should be included in style guides for the language. If the Java team had specified a standard convention for switches when it released the language (in the same way that it recommended initial capitals for class names and all capitals for constants), this small annoyance would not exist. The more a language can formalize small details, the easier it is to get things done.

But in an ideal world, even this solution is insufficient. I strongly believe that the abundance of Java style guides is itself a limitation. I'd far prefer that there be one consistent set of recommendations that was universally followed. For example, writing out definitive guidelines for the

location of opening braces, size of indents, tabs vs. spaces, how to stagger or not stagger if/else sequences, Javadoc's numerous formatting options, and so forth. Obviously, this would apply to higher-level concerns as well: fully expanded imports vs. wildcards, the sequence of import statements and variable declarations, and so on. By having a fixed set of guidelines, every Java listing would be consistent and not require re-examination to adjust to a given individual's or site's style.

Curiously enough, in a certain way, Java's initial appearance on the scene addressed what, at the time, was a tremendous laxity in language that made some tasks exceedingly tedious. The principal language before Java was C. It was purposely designed from a radically nonprescriptive perspective. Even today, after many rounds of standardization, C has numerous places where behavior is undefined or left up to the implementation to define. In the mid-'90s when Java first appeared, C was far looser. An integer could be more or less anything the compiler defined it to be, with—if I recall correctly—a minimum of 16 bits of width. 16, 32, and 64 bits were all legitimate implementations of an integer. As a result, porting C from one platform to another was extraordinarily tedious. Java solved these problems. Data items had fixed sizes across platforms, and code could be run on multiple platforms without modification.

C's lack of standardization caused so much pointless activity that when the original team from AT&T Bell Labs developed a new language, Go, they chose a highly prescriptive implementation. There is one formalized coding style for Go, and all code is expected to use that style. A code formatter is bundled with the Go distribution. In the language itself, there are additional constraints. For example, the executable code after any `if` statement must be enclosed in braces, even it if contains only a single line. Many other conventional items are defined by what is known as "idiomatic Go." The happy result is that all Go code looks the same. Reading and writing it is easy.

Most modern development organizations prescribe their own "house style" for code, which is frequently enforced in code reviews. But these styles conflict with each other for lack of a single, unified set of conventions.

The lack of standardization of details during the last few years has been an issue in Java in small but annoying ways, beyond command-line syntax. For example, the three variants of the annotation for indicating a field should not be null: `@NonNull`, `@Nonnull`, and `@NotNull`. The first of these was used by Checkstyle and FindBugs, the last of them by Java EE 6 and the IntelliJ IDE. The result was that if you coded in one environment and moved to a different development environment, you had to change your code or your toolchain to get your expected level of null-checking. This is, of course, the exact antithesis of Java's vaunted portability, when switching IDEs is enough to make code behave differently. Fortunately, Java 8's use of the Checker framework has now consolidated the convention around `@NonNull`.

The convenience and benefits of such strictures that ensure uniform syntax are widely recognized. This is evident in the choice of most modern development organizations to prescribe their own "house style" for code, which is frequently enforced in code reviews, so that all developers use the same conventions. But these styles conflict with each other for lack of a single, unified set of conventions.

If the world were all Java, I think it would not be too onerous to put up with the differences, although the time lost in doing so is lost for no good reason.

But in an increasingly polyglot world in which other languages (JavaScript, HTML, and so on) play significant roles, the lack of enforced coding standards in Java and especially across those languages combine to create a sustained and pointless drag on productivity.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

Coming in November: our special issue on JUnit 5.

JULY/AUGUST 2016

The Limitations of JSON-P

The article by David Delabassée titled "Using the Java APIs for JSON Processing" in the July/August 2016 issue does not mention significant problems with the JSON-P standard, which is poorly designed, and the reference implementation, which is poorly implemented.

I believe the standard is poorly designed because neither of the two APIs in the standard support parsing and serialization of plain old Java objects (POJOs) to and from JSON, the natural fit for an object-oriented language.

JSON parsers and serializers written with the JSON-P API are too big, require too many API calls, and are too fragile, requiring extensive changes if the structure of the JSON to be processed changes.

I believe the reference implementation is poorly implemented because in my use case, converting thousands of Java objects to JSON strings of 500 to 50,000 characters each, a program using the Jackson API's object model converted a real data set to JSON an astounding 10 times faster than one using the object model API in the JSON-P reference implementation. The presumably more efficient streaming API in JSON-P would have required unmaintainable code in which the API calls to open JSON arrays and objects were often widely separated from the API calls to close them.

So, I believe you should discuss alternative JSON processing libraries such as Jackson and GSON.

—Jeffrey S. Mayo
Norcross, Georgia

David Delabassée responds: "In the article, I discuss marshaling: 'Note that binding (that is, marshaling of Java objects to JSON documents and vice versa) will be addressed in a related API, the Java API for JSON Binding (JSON-B), which is currently being defined in JSR 367.'"

The editors add: "JSON-P is part of the Java EE standard and has an elegant parsing interface; in addition, it's fast. When JSON-B is finalized, we'll cover it in Java Magazine *as well. Jackson and GSON both represent good choices for developers needing JSON serialization now."*

The Practice of Small Classes

With respect to your editorial in the July/August issue, "The Problem of Writing Small Classes," I second your idea and argumentation about writing small classes in any "normal" code project. But when it comes to web development (servlets) or when writing a RESTful API, I always find it quite painful to externalize lots of pure (RESTful) logic just to stay within my own limits.

Just as an example, I have a REST resource class called `ItemResource`. I define several methods that make use of different media types and define several request mappings and request methods (using Spring, but it would be the same with pure Java EE).

So, alone with the imports, injections, method definitions, annotations, and logging plus the pure calls to business (Entity) classes and some Java 8 stream processing (mostly one-liners), I easily reach 200 lines of code or more.

I start to believe in the best of both worlds—trying to keep it simple and small but still binding semantic units together. A class should still be a class, and if we divide this more and more into smaller parts we could end up with one-method classes or endless delegations to deeper classes.

I tend to not limit myself to a hard number of lines, but take other measures in metrics plugins such as method complexity and try to take the separation of concerns seriously.

—Alex Hepp
Germany

Why Monospaced Fonts?

In your editorial in the July/August issue, you mention that the code font is under discussion. Why not use a nice sans-serif proportional font like Tahoma or any other open font? Most important: Do syntax highlighting!

Monospaced fonts for modern code samples are completely old-fashioned and totally horrible. Nobody needs to have columns vertically aligned anymore as was useful in COBOL and Fortran times.

The advantage of syntax highlighting is obvious. It is so much easier to see keywords, constants, comments, and so on with little effort. Which scheme is used for colors, italics, and font weight is highly subjective, of course.

—Hubert Kauker
Germany

Andrew Binstock responds: "I can see no possible benefit from giving up monospaced fonts for coding. I would note that every reputable website and publisher uses monospace. In addition, so does just about every developer I know—by which I mean every developer whose code I've seen.

"Syntax highlighting is not ideal. The first reason is that every reader has a different preferred scheme. Moreover, given our generally short printed listings (with downloadable code available for the entire codebase), the benefit would be small while the effort and cost required would be significant."

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

//events /

Devoxx Belgium *NOVEMBER 7–11*
ANTWERP, BELGIUM
Devoxx Belgium is dedicated to deep learning this year, touching on topics such as quantum computing and machine learning. Tracks will cover methodology; Java language skills; cloud, containers, and infrastructure; server-side Java; and more. An estimated 3,500 developers will choose from more than 200 sessions, including one-hour conference talks, three-hour hands-on labs, informal presentations, a program for kids, and more.

PHOTOGRAPH BY ERIC HUYBRECHTS/FLICKR

Desert Code Camp
OCTOBER 8
CHANDLER, ARIZONA
Desert Code Camp is a free, developer-based conference built on community content that promises "No Fluff: Only Code." Topics scheduled for the Java track include bots, Amazon Echo, and language user interfaces; getting started with Apache Kafka; and microservices with Spring Boot.

JAX London
OCTOBER 10 AND 13, WORKSHOPS
OCTOBER 11–12, CONFERENCE
LONDON, ENGLAND
JAX London is a four-day event for cutting-edge software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps. Speakers include Klara Ward, Oracle's Java Mission Control principal developer, and Raoul-Gabriel Urma, coauthor of *Java 8 in Action*.

JavaDay Kiev
OCTOBER 14–15
KIEV, UKRAINE
JavaDay Kiev will feature more than 50 sessions on topics ranging from the core JVM platform and Java SE (Java 8) to JVM languages to big data and NoSQL. Five keynotes are slated, and more than 1,000 developers are expected to attend this year's event. Juergen Hoeller, cofounder and project lead of the open source Spring Framework, is scheduled to speak.

O'Reilly Software Architecture Conference
OCTOBER 18–19, TRAINING
OCTOBER 19–21, TUTORIALS AND CONFERENCE
LONDON, ENGLAND
This year, the O'Reilly Software Architecture Conference is exploring evolutionary architecture to reflect the broadening of the field, encompassing disciplines such as DevOps. Topics will include strategies for meeting business goals, developing leadership skills, and making the concep-

tual jump from software developer to architect. Java-specific highlights include a session devoted to fault-tolerant programming with Hystrix and led by Pivotal's Matt Stine, a cloud-native Java primer with Pivotal's Josh Long, and a presentation on best practices in the use of containers by Red Hat's Mark Little.

Java Forum Nord

OCTOBER 20
HANOVER, GERMANY

Java Forum Nord is a one-day, noncommercial conference in northern Germany for Java developers and decision-makers. With more than 25 presentations in parallel tracks and a diverse program, the event also provides networking opportunities. (No English page available.)

VOXXED Days Thessaloniki

OCTOBER 21
THESSALONIKI, GREECE

The inaugural VOXXED Days event in Thessaloniki is a developer conference that promises interesting speakers, core developers of popular open source technologies, and professionals willing to share their knowledge and experiences. Former Oracle Technology Evangelist Simon Ritter is scheduled to present "JDK 9: Big Changes to Make Java Smaller."

Devoxx Morocco

NOVEMBER 1, UNIVERSITY DAY
NOVEMBER 2–3, CONFERENCE
CASABLANCA, MOROCCO

Devoxx Morocco is the place to go for learning, networking, and sharing experiences about Java, related technologies, and software craftsmanship. This year, more than 1,500 developers are expected, 150 sessions are slated, and activities are planned for children ages 9 to 15.

QCon

NOVEMBER 7–9, CONFERENCE
NOVEMBER 10–11, WORKSHOPS
SAN FRANCISCO, CALIFORNIA

Billed as a practitioner-driven software conference, QCon is designed for technical team leads, architects, engineering directors, and project managers. Tracks include an overview of Java 9 hosted by QCon Chair Wes Reisz that will focus on prepping for JDK 9 and cover concurrency updates, HTTP/2, unified logging, G1 garbage collector, Scala, and Reactive Streams.

W-JAX

NOVEMBER 7 AND 11, WORKSHOPS
NOVEMBER 8–10, CONFERENCE
MUNICH, GERMANY

The W-JAX conference is focused on Java, architecture, and software innovation. More than 160 presentations on technologies and languages—ranging from Java, Scala, and Android to web programming, agile development models, and DevOps—are planned. (No English page available.)

O'Reilly Software Architecture Conference

NOVEMBER 13–14, TRAINING
NOVEMBER 14–16, TUTORIALS AND CONFERENCE
SAN FRANCISCO, CALIFORNIA

Like its European counterpart, the California version of this conference will explore evolutionary architecture. Presentations of interest to Java developers include "An Introduction to Reactive Applications, Reactive Streams, and Options for the JVM" with ThirdChannel's Stephen Pember, and "Clone Clone Make: A Better

PHOTOGRAPH BY BENBENW/FLICKR

Way to Build," led by IBM's Dan Heidinga.

Topconf
NOVEMBER 15, WORKSHOPS
NOVEMBER 16–18, CONFERENCE
TALLIN, ESTONIA
Java and the JVM, reactive architectures, sustainable development, and big data highlight this year's conference. Couchbase Developer Advocate Laurent Doguin will present a practical example of RxJava, and MATHEMA Software Senior Consultant Thomas Künneth will discuss current approaches to Java on mobile devices.

Java Enterprise Summit
NOVEMBER 28–30
BERLIN, GERMANY
What does a state-of-the-art enterprise Java application look like? Which APIs are useful? What are the roles of various web and JavaScript frameworks? And how important is standard Java EE today? These and many other questions will be discussed at this year's Java Enterprise Summit. A large training event is held concurrently with the Micro Services Summit, hosting 24 power workshops with well-known German microservices and enterprise Java experts. (No English page available.)

ConFoo
DECEMBER 5–7
VANCOUVER, CANADA
This multitechnology conference for web developers features sessions on Java and JVM. Scheduled topics include Java 9; caching; machine-learning models with Java- and Spark-based tools; Docker and Java; and writing better streams with Java 8.

Special Note: Event Cancellation

QCon Rio
OCTOBER 5–7
RIO DE JANEIRO, BRAZIL
The organizers report: "Faced with an unstable political and economic environment . . . we considered it prudent to cancel our edition of QCon 2016. We emphasize that this decision does not affect the preparation of QCon São Paulo 2017."

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event four months in advance at javamag_us@oracle.com.

//event recap /

JVM Language Summit 2016

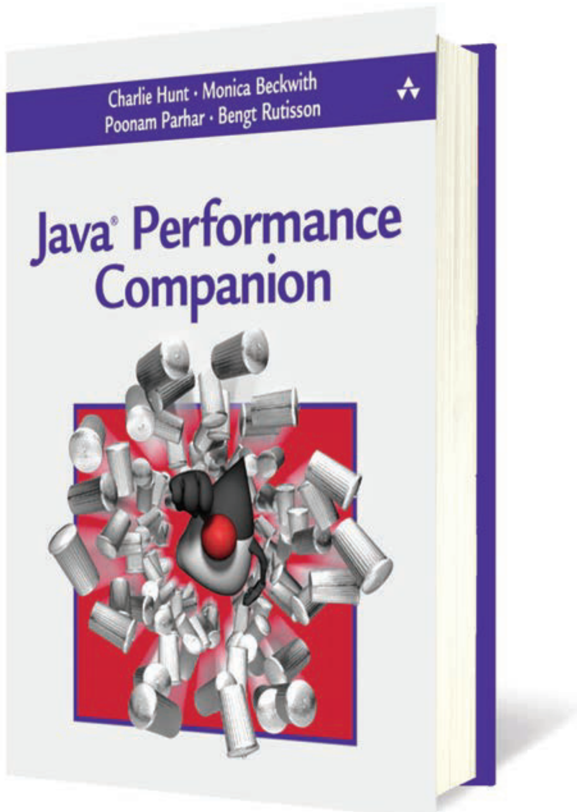
For the past nine years, Oracle (and earlier, Sun) has been sponsoring a small gathering of Java experts whose primary work is on the JVM. The roughly 150 attendees gather to discuss the JVM and JVM languages.

Unlike most conferences, at this summit there is only a single track, attended by all the participants. Over three days, many sessions of surpassing interest are presented. All of them are recorded. The videos for the 2016 summit are posted on YouTube. This year's summit in early August saw particular focus on Java 9 and post–Java 9 releases.

As for JVM languages, you can find videos of sessions on implementation details and upcoming features for Kotlin and Scala. And there are sessions that focus on standalone technical topics: immutable collections, bytecodes, use of Intel vector instructions, and so forth.

While the sessions are technically demanding, you do not need deep knowledge of the JVM to follow along. However, as you'll see in the videos, you do need some fluency with the terminology of JVM functions and features.

The conference is held in July or August at Oracle's campus in Santa Clara, California. While attendance is intentionally kept low, developers working on JVM topics can apply to attend on the conference's home page. Attendance is generally opened 60 days before the next summit begins. The home page also includes links to the videos of sessions from previous summits. Enjoy the deep dives! —*Andrew Binstock*



JAVA PERFORMANCE COMPANION

By Charlie Hunt, Monica Beckwith, et al.
Addison-Wesley

This book's title—in which *Companion* is the operative word—indicates its primary mission: a supplement to the highly regarded classic on JVM performance tuning, *Java Performance*, by Charlie Hunt and Binu John. However, that book first appeared toward the end of 2011 and has not been updated by a second edition. So, this new volume provides information on the central performance topic that came after 2011: the G1 garbage collector. While G1, which is a standard part of Java releases since JDK 7, update 4, has been designed to require only a few options (after which the garbage collector will sort out its ideal configuration), certain workloads can benefit from an admin's additional tuning through switches and runtime parameters.

The discussion of G1 starts with an introduction comparing garbage collectors, followed by a deep dive into its internals. And, finally, there are explanations (and recommendations) regarding the various configuration options that can be set by an admin. The writing is clear but does make occasional references to the original book. It's best to read this volume as an adjunct to that work, although for readers familiar with JVM innards, it can be read by itself.

The second half of the book is dedicated to the Java Serviceability Agent (SA), a little-known debugging utility in the JDK. The SA attaches to a running process and takes a complete snapshot, which it translates into human-readable format. It includes full stack and heap info, plus data on the running processes. In addition, it provides very detailed analysis of core dumps when programs lock up or suffer fatal exceptions. This data supplements that provided by jhat, the mainstay JDK analyzer of heap dump files.

Finally, an appendix reviews performance-related command-line options that have been added to the JVM since the original book appeared. Each switch enjoys several paragraphs of explanation and recommendations on its use.

Although a US$50 cover price for a 150-page book seems more than a tad on the expensive side, the information in this slim volume will be valuable enough to sites that need top performance that the price should represent no obstacle. With that in mind, I have no difficulty recommending this book unreservedly.

—Andrew Binstock

Programming Devices

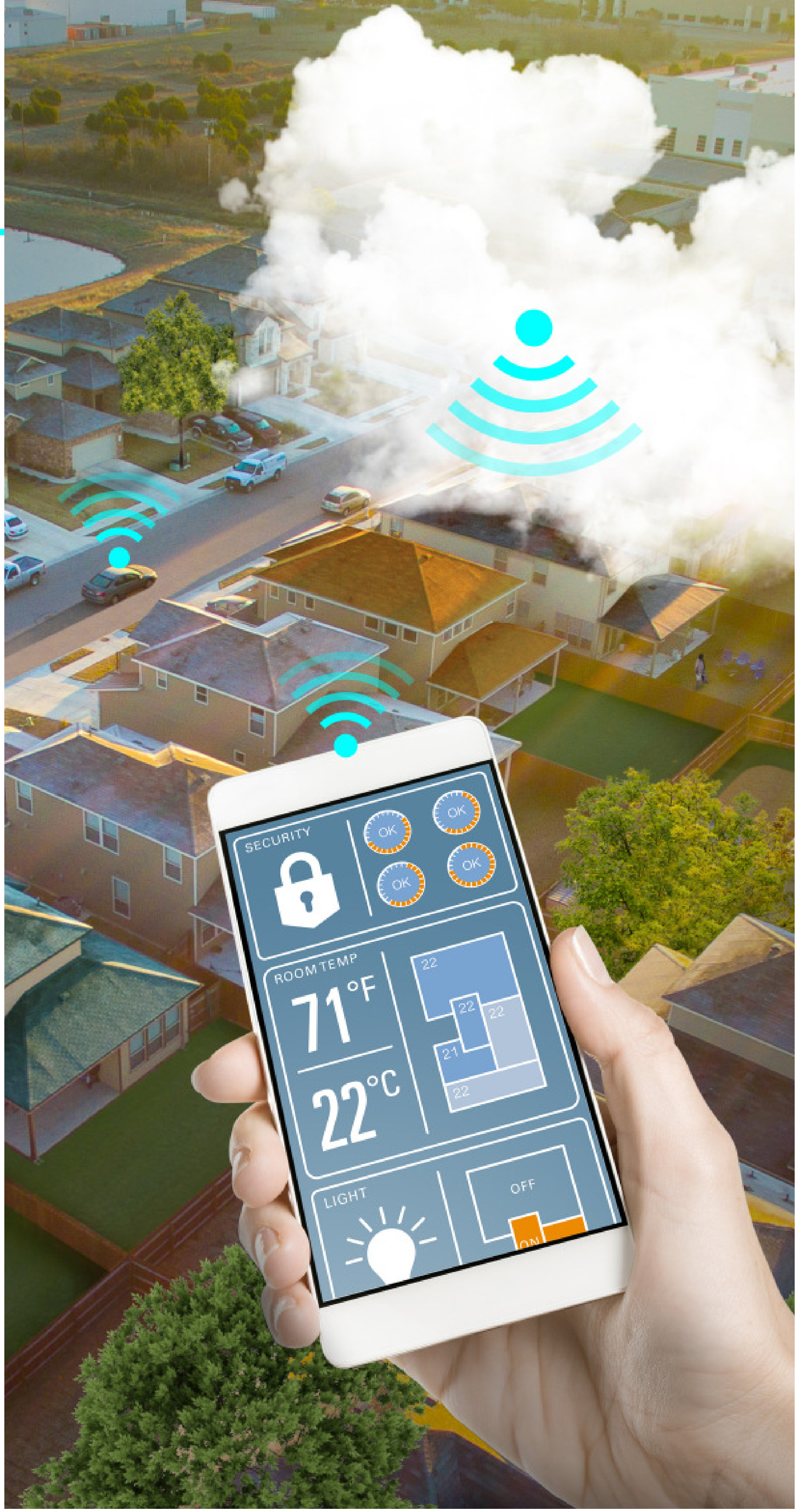
In this issue, our feature articles focus on how to program devices. The devices range from small, single-purpose hardware items to complex machines. When hooked together, typically via the cloud, they form the Internet of Things (IoT). Much of IoT programming consists of raw data processing: you gather data from the device, and you send it commands. The biggest challenges are accessing the device and knowing the command sequence that translates into specific actions. Learn how to do most of this, add a little back-end processing in the cloud, and you're most of the way there.

This issue has three articles that build on each other to show how this kind of programming is done in Java. The first article, by Gastón Hillar (page 14), shows the basics of accessing a device—in this case, an inexpensive Intel-based IoT board—and using its features by querying sensors and turning on various pins that light an LED. If you're not terribly familiar with Java access to hardware, this is the article to start with. It's an ideal hobbyist project: instructive and useful, and it can be completed in a single afternoon.

The second article (page 24) demonstrates how to control a CNC router by sending commands to it from a Raspberry Pi board. CNC routers are devices that shape hardware items by grinding down or cutting a piece of material into the right shape. They're like 3-D printers except that they remove material from a block rather than iteratively build up the product from new material. As this article shows, once a connection to the device is established, the primary task is getting data from the device and sending it new commands. The Java code for managing these two activities is remarkably straightforward. The author, Stephen Chin, wrote a similar article in our May/June 2015 issue explaining how to interact with an electronic scale from a Raspberry Pi.

For the IoT to become a reality, devices need to share data and typically will do so by funneling the data from multiple devices to a cloud instance. Our third article on IoT (page 32) shows how this works on a cloud platform as a service custom-made for such data and for controlling devices remotely.

In addition, you'll find our regular assortment of Java articles: in-depth tutorials, explorations of JVM languages, book reviews, letters we've received, and of course, our Java quiz.



ART BY I-HUA CHEN

GASTÓN HILLAR

Interacting with Sensors on Intel's x86 Galileo Board

Capturing data and responding to it are the heart and soul of IoT. Here's how it works with Java and an inexpensive board that uses an x86 processor.

In this article, I develop and explain a project that measures ambient light and dims a red, green, blue (RGB) LED in response. This way, you can see how to interact with analog inputs and pulse width modulation (PWM) outputs using Java SE 8. I use the latest versions of the Intel IoT Development Kit, which includes support for Java. I take advantage of both the upm and mraa libraries, which provide high-level interfaces for controlling the Intel Galileo Gen 2 board—an Arduino-compliant x86 chip based on the Pentium architecture. I also show how to control wired electronic components, sensors, and actuators.

Prerequisites

The Intel Galileo Gen 2 board boots the Yocto Poky Linux image, which is the latest version of the Intel Galileo microSD card Linux operating system image. You can download the latest version of the Yocto Poky image for Galileo from the Intel website.

For development, I use the Intel System Studio IoT Edition software. It is an Eclipse-based IDE that makes it easy to create a new IoT project with Java as the main programming language. I also use the latest available versions of both the mraa and upm libraries. The mraa library is a low-level skeleton library for communication on Linux platforms, and upm is a set of libraries for interacting with sensors and actuators. Both are part of the Intel IoT Development Kit. You can download Intel System Studio IoT Edition from Intel's IoT site.

One of the main problems you can face when you start a new IoT project with Java is that the versions for the upm and mraa libraries included in Intel System Studio IoT Edition must match the versions of these libraries installed in the version of Yocto Poky Linux that is running on the Intel Galileo Gen 2 board. Unfortunately, I had many problems when using the synchronization features included in the IDE and, therefore, I suggest first updating the libraries in the IDE and then in the board as separate operations. Select `Intel(R) IoT | Libraries Update…` in the main menu for the IDE and follow the steps to install the latest available versions for the libraries.

Once the board has finished the boot process with the microSD card Linux operating system image and is connected to your LAN through the Ethernet port, the DHCP server will have assigned the board an IP address. You will need this IP address later. There are many ways for you to retrieve the assigned address. For example, if you have access to the web interface for your DHCP server, you can use it to retrieve the IP assigned to the device with the MAC address that matches the one printed on a label in the board.

You can also discover the board and its services on the LAN automatically through the zero-configuration network-

ing implementation. In Windows, you can use the free Bonjour Browser for Windows. In OS X, you can use the free Bonjour Browser for OS X.

Both applications display the available Bonjour services, and you just need to pay attention to those named `galileo`. The IP address displayed for the SSH service named `galileo` is the one you can use to establish a connection with the board.

Then, use your favorite SSH terminal tool to connect to Yocto Linux running on the Intel Galileo Gen 2 board by using the previously retrieved IP address, the `root` user, and a blank password. Then, execute the following commands in the SSH terminal:

```
opkg update
opkg install mraa
opkg install upm
```

Then, click `Create an IoT Project` and select `Java Project` in the IDE. I'll use AmbientLightAndLed as the project name. Enter the connection name and the IP address for your Intel Galileo Gen 2 board. This way, the IDE will provide you with an environment that will upload and run the project to the specified board. In the console, you will see the output generated by the project, and you can run commands in a terminal. However, as always, once you learn how things work, you will find it easier to generate your own scripts and run your own commands to upload the generated JAR file and run it. The IDE provides an integrated debugging experience that is extremely useful when you start working with the mraa and upm libraries in Java.

Wiring the Electronic Components

A *photoresistor* is an electronic component also known as a light-dependent resistor (LDR) or photocell. It is not the best component for sensing ambient light with high levels of accuracy. However, the component is useful for determining whether the environment is dark, and this example doesn't have problems with latencies (that is, delays in reporting the dimming of light). Just make sure you take into account accuracy and latency if you need to sense ambient light in a more complex project.

The photoresistor is a variable resistor whose resistance value changes based on the ambient light intensity. When the ambient light intensity increases, the resistance of the photoresistor decreases and vice versa.

The board makes it possible to read voltage values through the analog input pins. I use a voltage divider configuration that includes the photoresistor as one of its two resistors. The voltage divider outputs a high voltage value when the photoresistor receives a high amount of light, and it outputs a low voltage value when the photoresistor receives a small amount of light or no light at all.

The Intel Galileo Gen 2 board allows you to use only six of the digital I/O pins as PWM output pins. These six pins are labeled with a tilde symbol (~), which reflects a prefix to the number on the board.

I use the analog pin labeled A0 to connect the positive side (+) of the voltage divider that includes the photoresistor. In addition, I use the following PWM-enabled pins to control the brightness level of an RGB LED:

- Pin ~6 is connected to the anode pin for the red component of the RGB LED.
- Pin ~5 is connected to the anode pin for the green component of the RGB LED.
- Pin ~3 is connected to the anode pin for the blue component of the RGB LED.

You need the following parts to work with this example:

- A photoresistor.
- A 10,000Ω (10 kΩ) resistor with 5 percent tolerance. The color bands for the resistor are brown, black, orange, and gold.
- A common cathode 5 mm RGB LED.

- Three 270Ω resistors with 5 percent tolerance. The color bands for the resistor are red, violet, brown, and gold.

Figure 1 shows the previously mentioned electronic components connected to a breadboard, the necessary wirings, and the wirings from the Intel Galileo Gen 2 board to the breadboard. (I created the original diagram with the popular Fritzing multiplatform application. The Fritzing file is included in the code bundle related to this article, which is available in the *Java Magazine* download area.

There are three PWM-capable GPIO (general-purpose input/output) pins: ~6, ~5, and ~3. Each of them is connected to a 270Ω resistor and wired to an anode pin for each LED color. The common cathode is connected to ground (GND). The analog input pin labeled A0 is connected to the voltage divider built from the photoresistor and a 10 kΩ resistor with 5 percent tolerance. The photoresistor is wired to the IOREF pin. I am using the board's default configuration and, therefore, the IOREF voltage is 5V. The 10 kΩ resistor is wired to GND.

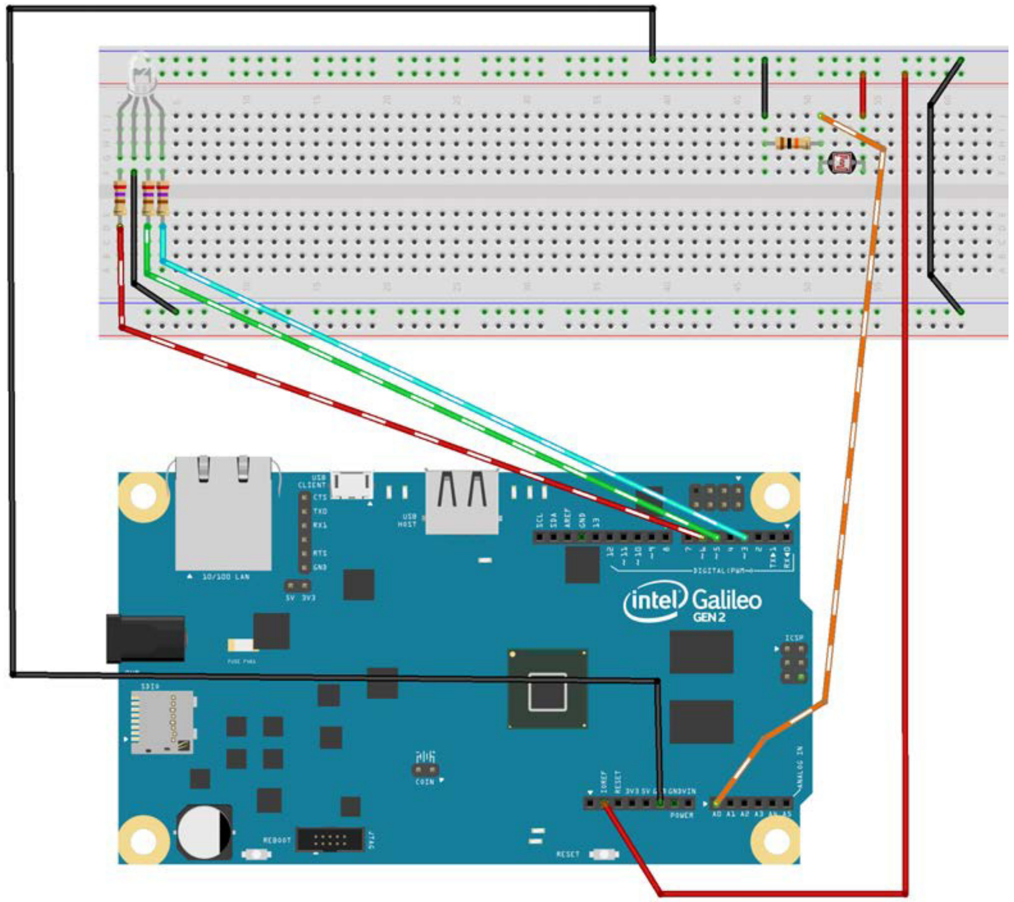


Figure 1. The electronic components connected to a breadboard and wired to the Intel Galileo Gen 2 board

Coding a Class to Dim an LED by Using PWM

Once you finish all the necessary wiring, you need to write Java code to determine whether you are in a dark environment and then control the brightness of the three colors of the RGB LED based on the ambient light value. The code reads the result of converting a resistance value into a voltage, and then transforms this analog value into its digital representation. The code maps the digital value to a voltage value, and then it maps this voltage value to a darkness or ambient light measurement value.

In order to keep the code easier to read and understand, the next Java classes I explain won't perform checks on the results of each operation performed. However, in a final version of the example, you should check the result of each call to a method of an instance of the different mraa classes and make sure that the returned value is equal to `mraa.Result.SUCCESS`.

I am going to create the following three classes:

- `VariableBrightnessLed`: This class represents an LED connected to the board, and it will allow me to control its brightness level.
- `VoltageInput`: This class represents a voltage source connected to an analog input pin in the board, and it will allow me to map the raw values read from the analog input into voltage values.
- `SimpleLightSensor`: This class represents a light sensor, and it will allow me to transform a voltage value, measured with a `VoltageInput` instance, into a light measurement and description.

Then, I will create a `BoardManager` class that creates instances

of the VariableBrightnessLed and SimpleLightSensor classes, and will interact with them. Finally, I will create a master class, called AmbientLightAndLed, which coordinates the operations of the previously explained classes.

First, I create a new VariableBrightnessLed class that represents an LED connected to the board that can have a brightness level from 0 to 255 inclusive. The following code lines show the import statements that I use for all the classes and the code for the new class:

```
import mraa.Aio;
import mraa.Pwm;

class VariableBrightnessLed {
    private final int gpioPin;
    private final String name;
    private final Pwm pwm;
    private int brightnessLevel;


    public VariableBrightnessLed(int gpioPin,
        String name) {
        this.name = name;
        this.gpioPin = gpioPin;
        this.pwm = new Pwm(gpioPin);
        this.pwm.period_us(700);
        this.pwm.enable(true);
        // Set the initial brightness level to 0
        this.setBrightnessLevel(0);
    }

    public void setBrightnessLevel(
        int brightnessLevel) {
        int validBrightnessLevel = brightnessLevel;
        if (validBrightnessLevel > 255) {
            validBrightnessLevel = 255;
        } else if (validBrightnessLevel < 0){
            validBrightnessLevel = 0;
        }
        float convertedLevel =
            validBrightnessLevel / 255f;
        this.pwm.write(convertedLevel);
        this.brightnessLevel = validBrightnessLevel;
        System.out.format(
            "%s LED connected to PWM Pin #%d " +
            "set to brightness level %d.%n",
            this.name,
            this.gpioPin,
            validBrightnessLevel);
    }

    public int getBrightnessLevel() {
        return this.brightnessLevel;
    }

    public int getGpioPin() {
        return this.gpioPin;
    }
}
```

When I create an instance of the VariableBrightnessLed class, it is necessary to specify the GPIO pin number to which the LED is connected in the gpioPin int argument and a name for the LED in the name String argument. The constructor creates a new mraa.Pwm instance with the received gpioPin as its pin argument, saves its reference in the pwm field, and calls its period_us method to configure the PWM period to 700 microseconds (700 µs). This way, the output duty cycle will determine the percentage of the 700 µs period during which the signal is in the on state.

For example, a 0.10 (10 percent) output duty cycle means that the signal is on during 70 µs of the 700 µs period.

The class declares a method that translates a brightness level from 0 to 255 (inclusive) into the appropriate output duty-cycle value for the PWM pin.

Then, the constructor calls the pwm.enable method with true as a parameter to set the enable status of the PWM-enabled pin and allow the code to start setting the output duty-cycle percentage for the PWM pin with calls to the pwm.write method.

Finally, the constructor calls the setBrightnessLevel method with 0 as the value for the brightnessLevel argument. This way, the constructor sets the brightness level to 0 for the LED wired to the specified pin number and turns off the LED. Specifically, the call turns off a specific color component of the RGB LED.

The class declares a setBrightnessLevel method that translates a brightness level from 0 to 255 (inclusive) into the appropriate output duty-cycle value for the PWM pin. The method receives a brightness level int value in the brightnessLevel argument.

First, the code makes sure that the brightness level is a value between 0 and 255 (inclusive). If the value is out of range, the code uses either the lower-level or the upper-level value and saves it in the validBrightnessLevel local variable.

Then, the code calculates the required output duty-cycle percentage for the PWM pin to represent the brightness level as a float value between 1.0f and 0.0f (100 percent and 0 percent). It is necessary to divide the valid brightness level (validBrightnessLevel) by 255f. The code saves the value in the convertedLevel variable. The next line calls the this.pwm.write method with the convertedLevel variable for the percentage argument and sets the output duty cycle for the pin configured as the PWM output to convertedLevel.

Finally, the code saves the valid brightness level in the brightnessLevel field, which is read-only accessible through the getBrightnessLevel method. The last line prints details about the brightness level set to the LED identified with a name and wired to a specific pin number. The line is printed with System.out.format, and it is possible to see the output when you run the generated JAR file through the IDE or by running commands through SSH on Yocto Linux running on the board. I'll dive deep on the benefits of printing useful information later.

Measuring Ambient Light via Analog Input

Now, I create a new VoltageInput class that represents a voltage source connected to an analog input pin on the board. The following lines show the code for this new class:

```
class VoltageInput {
    private final int analogPin;
    private final Aio aio;

    public VoltageInput(int analogPin) {
        this.analogPin = analogPin;
        this.aio = new Aio(analogPin);
        // Configure the ADC
        // (short for Analog-to-Digital Converter)
        // resolution to 12 bits (0 to 4095)
        this.aio.setBit(12);
    }

    public float getVoltage() {
        long rawValue = this.aio.read();
        float voltageValue =
            rawValue / 4095f * 5f;
        return voltageValue;
    }
```

```
    public int getAnalogPin() {
        return analogPin;
    }
}
```

When I create an instance of the VoltageInput class, it is necessary to specify, in the analogPin int argument, the analog pin number to which the voltage source is connected. The constructor saves the received analog pin number in the analogPin field, which is read-only accessible through the getAnalogPin method. Then, the constructor creates a new mraa.Aio instance with the received analogPin as its pin argument, saves its reference in the aio field, and calls its setBit method to configure the analog-to-digital converter (ADC) resolution to 12 bits. This way, the ADC will provide 4,096 possible values ($2^{12} = 4096$) to represent from 0V to 5V. A 0 value read from the ADC represents 0V, and 4095 means 5V.

It is necessary to apply a linear function to convert the raw values read from the analog pin and map them to the corresponding input voltage values. Thus, the code multiplies the raw value read from the analog pin by 5 and divides it by 4095 to obtain the input voltage value from the raw value. As I am using 12 bits of resolution, the detected values will have a step of 5V / 4095 = 0.001220012V, that is, approximately 1.22 millivolts.

The VoltageInput class declares a getVoltage method that calls the this.aio.read method to retrieve the raw value from the analog pin and saves it in the rawValue long variable. The code saves the value in this variable to make it easy to debug and understand how the code works. Then, the method calculates the result of dividing rawValue by 4,095 and multiplying it by 5, and saves it to the float voltageValue variable. Finally, the method returns the value of the voltageValue variable. This way, the method returns the voltage value, converted from the raw value retrieved by the this.aio.read method.

I have a class that allows me to retrieve a voltage value from a voltage source. Now, I will create a new SimpleLightSensor class that represents the photoresistor, which is included in the voltage divider and wired to an analog pin of the board. The new class uses the previously coded VoltageInput class that reads and transforms an analog input. The new class allows me to transform a voltage value into a light measurement and description. The following lines show the code for this new class:

```
class SimpleLightSensor {
    // Light-level descriptions
    public static final String
        LL_EXTREMELY_DARK = "Extremely dark";
    public static final String
        LL_VERY_DARK = "Very dark";
    public static final String
        LL_JUST_DARK = "Just dark";
    public static final String
        LL_SUNNY_DAY = "Like a sunny day";
    // Maximum voltages that determine the light level
    private static final float
        EXTREMELY_DARK = 2.1f;
    private static final float
        VERY_DARK = 3.1f;
    private static final float
        JUST_DARK = 4.05f;

    private final VoltageInput voltageInput;
    private float measuredVoltage = 0f;
    private String lightLevel =
        SimpleLightSensor.LL_SUNNY_DAY;

    public SimpleLightSensor(int analogPin) {
        this.voltageInput =
            new VoltageInput(analogPin);
```

```
        this.measureLight();
    }

    public void measureLight() {
        this.measuredVoltage =
            this.voltageInput.getVoltage();
        if (this.measuredVoltage <
            SimpleLightSensor.EXTREMELY_DARK) {
            this.lightLevel =
                SimpleLightSensor.LL_EXTREMELY_DARK;
        } else if (this.measuredVoltage <
            SimpleLightSensor.VERY_DARK) {
            this.lightLevel =
                SimpleLightSensor.LL_VERY_DARK;
        } else if (this.measuredVoltage <
            SimpleLightSensor.JUST_DARK) {
            this.lightLevel =
                SimpleLightSensor.LL_JUST_DARK;
        } else {
            this.lightLevel =
                SimpleLightSensor.LL_SUNNY_DAY;
        }
    }

    public String getLightLevel() {
        return this.lightLevel;
    }
}
```

The class defines the following three class fields, which specify the maximum voltage values that determine each light level.

- `EXTREMELY_DARK`: 2.1V
- `VERY_DARK`: 3.1V
- `JUST_DARK`: 4.05V

If the retrieved voltage is lower than 2.1V, the environment is extremely dark. If the retrieved voltage is lower than 3V, the environment is very dark. If the retrieved voltage is lower than 4.05V, the environment is just barely dark. These values work with a specific photoresistor; you might have to check the voltage values that determine specific environments in your own configuration. It is necessary to change only the values of the class fields.

The new class allows me to transform a voltage value into a light measurement and description.

The main goal for the `SimpleLightSensor` class is to convert a quantitative value (a voltage value) into a qualitative value (an ambient light description). The class declares the following four class fields with the descriptions for the light levels:

- `LL_EXTREMELY_DARK`
- `LL_VERY_DARK`
- `LL_JUST_DARK`
- `LL_SUNNY_DAY`

When I create an instance of the `SimpleLightSensor` class, I need to specify in the `analogPin` argument, which is the analog pin number to which the voltage divider that includes the photoresistor is connected. The constructor creates a new `VoltageInput` instance with the received `analogPin` argument and saves its reference in the `voltageInput` field. The next line calls the `measureLight` method that transforms the voltage value retrieved with the `VoltageInput` instance (`this.voltageInput`) into a description of the light level.

The class declares the `measureLight` method, which saves the voltage value retrieved by calling the `this.voltageInput.getVoltage` method in the `measuredVoltage` field. Then, the next lines use the previously explained class fields to determine whether the value of the `measuredVoltage` field is lower than the maximum voltages that determine each light level. The code sets the appropriate value for the `lightLevel` field according to the measured voltage value. Then, it is

possible to access the light level description by calling the `getLightLevel` method.

Controlling One Input and Three Outputs with a Board Manager

Now, I will create a new `BoardManager` class that creates an instance of the previously coded `SimpleLightSensor` class and three instances of the `VariableBrightnessLed` class. This way, there is one instance of the `VariableBrightnessLed` class for each color component of the RGB LED. The class fires actions when the ambient light changes. Specifically, the class adjusts the brightness for the three color components of the RGB LED based on the measured ambient light. The following lines show the code for this new class:

```
class BoardManager {
    public final SimpleLightSensor lightSensor;
    public final VariableBrightnessLed redLed;
    public final VariableBrightnessLed greenLed;
    public final VariableBrightnessLed blueLed;

    public BoardManager() {
        this.lightSensor =
            new SimpleLightSensor(0);
        this.redLed =
            new VariableBrightnessLed(6, "Red");
        this.greenLed =
            new VariableBrightnessLed(5, "Green");
        this.blueLed =
            new VariableBrightnessLed(3, "Blue");
    }

    public void setRGBBrightnessLevel(int value) {
        this.redLed.setBrightnessLevel(value);
        this.greenLed.setBrightnessLevel(value);
        this.blueLed.setBrightnessLevel(value);
    }

    public void updateLedsBasedOnLight() {
        String lightLevel =
            this.lightSensor.getLightLevel();
        switch (lightLevel) {
            case SimpleLightSensor.LL_EXTREMELY_DARK:
                this.setRGBBrightnessLevel(255);
                break;
            case SimpleLightSensor.LL_VERY_DARK:
                this.setRGBBrightnessLevel(128);
                break;
            case SimpleLightSensor.LL_JUST_DARK:
                this.setRGBBrightnessLevel(64);
                break;
            default:
                this.setRGBBrightnessLevel(0);
                break;
        }
    }
}
```

The `BoardManager` class declares the following four fields that the constructor initializes:

- `lightSensor`: An instance of `SimpleLightSensor`, which represents the photoresistor included in the voltage divider connected to the analog pin labeled A0.
- `redLed`: An instance of `VariableBrightnessLed`, which represents the red component of the RGB LED connected to the GPIO pin labeled ~6.
- `greenLed`: An instance of `VariableBrightnessLed`, which represents the green component of the RGB LED connected to the GPIO pin labeled ~5.
- `blueLed`: An instance of `VariableBrightnessLed`, which represents the blue component of the RGB LED connected to the GPIO pin labeled ~3.

The `setRGBBrightnessLevel` method calls the `setBrightnessLevel` method for the three `VariableBrightnessLed` instances with the `value` received as an argument. This way, the three color components of the RGB LED are set to the same brightness level through a single call.

The `updateLedsBasedOnLight` method retrieves the light-level description from the `SimpleLightSensor` instance and calls the previously explained `setRGBBrightnessLevel` method to set the brightness level for the three components of the RGB LED based on the measured light. If it is extremely dark, the brightness level is set to 255. If it is very dark, the brightness level is set to 128. If it is just dark, the brightness level is set to 64. Otherwise, the brightness level is set to 0, which means the RGB LED is completely off.

The main goal for the SimpleLightSensor class is to convert a quantitative value (a voltage value) into a qualitative value (an ambient light description).

Now, I will write code that uses the `BoardManager` class to measure the ambient light and set the brightness for the three color components of the RGB LED based on the measured ambient light. The following lines show the code for the new `AmbientLightAndLed` class:

```
public class AmbientLightAndLed {
    public static void main(String[] args) {
        String lastlightLevel = "";
        BoardManager board = new BoardManager();
        while (true) {
            board.lightSensor.measureLight();
            String newLightLevel =
                board.lightSensor.getLightLevel();
            if (newLightLevel != lastlightLevel) {
                // The measured light level has changed
                lastlightLevel = newLightLevel;
                System.out.format(
                    "Measured light level: %s%n",
                    newLightLevel);
                board.updateLedsBasedOnLight();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.err.format(
                    "Sleep interruption: %s",
                    e.toString());
            }
        }
    }
}
```

The class declares the `main` method that is executed when I upload and launch the project on the board. First, the method initializes the `lastLightLevel` local variable with an empty String and creates an instance of the `BoardManager` class named `board`. Then, the method runs a loop forever.

The loop calls the `board.lightSensor.measureLight` method to update the ambient light measurement. The next line saves the light-level description retrieved with the call to `board.lightSensor.getLightLevel` in the `newLightLevel` local variable. If the new light level is different from the last recorded ambient light level, the code updates the value for the `lastLightLevel` variable, prints the measured light level, and calls the `board.updateLedsBasedOnLight` method.

You can run the example and use a flashlight or your smartphone to move light over the photoresistor. You should see the printed messages and see that the RGB dims and finally turns off. After you reduce the light in the environment, the RGB LED increases its brightness.

```
Listening for transport dt_socket at address: 8009
Red LED connected to PWM Pin #6 set to brightness level 0.
Green LED connected to PWM Pin #5 set to brightness level 0.
Blue LED connected to PWM Pin #3 set to brightness level 0.
Measured light level: Extremely dark
Red LED connected to PWM Pin #6 set to brightness level 255.
Green LED connected to PWM Pin #5 set to brightness level 255.
Blue LED connected to PWM Pin #3 set to brightness level 255.
Measured light level: Extremely dark
Red LED connected to PWM Pin #6 set to brightness level 128.
Green LED connected to PWM Pin #5 set to brightness level 128.
Blue LED connected to PWM Pin #3 set to brightness level 128.
```

Figure 2. Console messages

Figure 2 is an example of the output shown in the console window of the IDE. The console window displays all the messages I printed, which makes it easy for me to understand what should happen with the components.

Conclusion

This simple example demonstrates that you can take advantage of Java SE 8 to create high-level code that interacts with IoT components on the Intel Galileo Gen 2 board. The mraa and upm libraries are regularly updated, and combining them with Java makes it easy to leverage your existing knowledge to undertake IoT projects that interact with different kinds of inputs, outputs, sensors, displays, and actuators. </article>

Gastón Hillar (@gastonhillar) has been working as a software architect with Java since its first release. He has 20 years of experience designing and developing software. He is the author of many books related to software development, hardware, electronics, and the Internet of Things.

learn more

[Yocto Poky Linux image from the Yocto Project]

[Intel System Studio IoT Edition User Guide for Java]

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 293: Revising the Format of Command-Line Options

This recent enhancement proposal suggests that the JDK tools that are run from the command line should share a common syntax for passed-in arguments. As the document points out, at present JDK tools don't share a common syntactical convention for switches. For example, the help command, which most readers will agree should be consistent across all tools, varies between `-?`, `-help`, and `--help`.

While JEP 293 primarily advocates uniformity in future command-line JDK products, it also suggests that a useful format to adopt would be the set of GNU syntax conventions, described in large part in getopt(3). These conventions use a single hyphen for single-letter options (which can be combined) and a double hyphen for the so-called "long" options. Although this choice is likely best because it's familiar to many users and libraries exist that already parse its syntax, it suffers from its own peculiarities (`-W` is reserved) and conflicts at points with the POSIX syntax. Of course, this kind of dialogue is precisely the point of these JEPs—to invite the community to share its views on potential changes to the JDK.

This proposal wisely doesn't advocate changing the options of existing tools, so no compatibility breakage is envisioned. Instead, it recommends that new releases all support the agreed-upon syntax, and, presumably, that existing tools eventually add duplicate switches to attain conformity.

STEPHEN CHIN

Raspberry Pi–Controlled CNC Router

Programming the Raspberry Pi to manage cutting, carving, and routing

CNC routers are very useful tools for creating physical objects by carving material out of blocks of wood, plastic, or even soft metals. (*CNC* stands for *computer numerical control* and simply indicates that the routers are directed by computers.) They are the opposite of a 3-D printer, which creates objects by adding material, but they are equally useful. Besides cutting out flat objects, by the careful removal of material in small layers you can carve complex 3-D geometries into durable materials.

Traditionally CNC routing was limited to large industrial machines, but modern desktop CNC routers are small enough to fit beside your computer. They are relatively quiet and have full enclosures to limit the spread of dust and shavings. They use motors similar to 3-D printer motors, so they have very high accuracy that can be used to create fine detail in carvings. And they are relatively easy to control, because they accept standard G-code instructions from an attached computer or microcontroller. G-code is a simple text-based language for describing low-level machine instructions that control a CNC router, 3-D printer, or other machinery. There are G-code commands to control coordinates, movement, rotation, and other machine functions.

In this article, I show you how to interface with the Nomad 883 Pro router produced by Carbide 3D. This router accepts G-code over a serial connection and uses a Grbl controller on an embedded Arduino board. If you are using a different printer there will most likely be different G-code initialization instructions and possibly a different controller board, so check your printer specifications.

You can find the complete code for the example I use in this article on GitHub.

Connecting to the Router

For sending data to the router, I use the UniversalGcode Sender project by Will Winder. It exposes a simple API for sending commands to an attached router that uses either a Grbl or TinyG controller.

To start, download the latest version of UniversalGcode Sender (version 1.0.9, as of the writing of this article) from the download page.

Then create a new project in your favorite IDE (I happen to be using NetBeans) and include the UniversalGcodeSender .jar file as a dependency in your project. You can create a new connection to the router as follows:

```
static GrblController grblController;
static final CutterListener listener =
    new CutterListener();
static String PORT_NAME = "/dev/ttyACM0";

public static void main(String[] args) throws Exception {
    // Send decimals with "." instead of ",":

    Locale.setDefault(Locale.ENGLISH);
    grblController = new GrblController();
    grblController.addListener(listener);
    grblController.setSingleStepMode(true);
    Boolean openCommPort =
```

```
        grblController.openCommPort(PORT_NAME, 115200);
    if (openCommPort != true) {
        throw new IllegalStateException(
            "Cannot open connection to the cutter");
    }
}
```

PORT_NAME is a constant that specifies the serial port your router is using, and 115200 is the baud rate at which your router communicates. To determine the port name of your router on Mac OS X or Linux, check the output of the dmesg command right after connecting the router. On Windows, you can use Device Manager to determine the port name by checking the Ports section for attached devices.

Note that I am using single-step mode to avoid errors on the Nomad 883 Pro. It prefers commands to be sent sequentially rather than having them be queued.

Once you have determined the correct port name and initiated a connection to the router, the next step is to initiate the router's homing and tool-measurement step. To initialize the Nomad 883 Pro, I execute the following G-code commands:

```
static final List<String> PROBE1 =
    Arrays.asList("G4P0.005", "M05", "G92.1",
          "G54", "G10 L2 P1 X0 Y0 Z0", "G21",
          "G49", "G90", "G10 L2 P1 X0 Y0 Z0",
          "G0 X-2.5 Z-5", "G0 Z-35.000",
          "G38.2Z-105 F800", "G4P0.005");

static final List<String> PROBE2 =
    Arrays.asList("G0 Z-70",
        "G38.2Z-182.675F200.0", "G4P0.005");

static final List<String> PROBE3 =
    Arrays.asList("G0 Z-5", "G0 X-5");
```

Create a new project in your favorite IDE (I happen to be using NetBeans), and include the UniversalGcodeSender.jar file as a dependency in your project.

PROBE1 initializes the coordinate system and moves to the probe location. Then it checks the probe length at a rate of 800 mm/m. PROBE2 repeats the probing at a slower, more accurate speed of 200 mm/m. Finally, PROBE3 moves the head away from the work surface and back to the home position.

Because the GrblController is designed to be asynchronous, issuing commands requires a little bit of work. To simplify this process, I created some wrapper commands that send commands and wait for them to finish. Using those wrapper methods, here is the initialization cycle:

```
waitForConnection();
homeAndWait();
sendSequenceAndWait(PROBE1);
sendSequenceAndWait(PROBE2);
sendSequenceAndWait(PROBE3);
```

Before you can issue commands, the initialization cycle of the router has to finish. For this, I wait until I get back the initialization messages from the CNC router:

```
private static void waitForConnection()
    throws InterruptedException {
        synchronized (listener) {
            while (!listener.connected) {
                listener.wait();
            }
        }
}
```

The implementation of `homeAndWait` calls the underlying `performHomingMethod` on the controller and then waits for the command to finish executing, as shown here:

```
static void homeAndWait() throws Exception {
    synchronized(listener) {
        listener.commandComplete = false;
        grblController.performHomingCycle();
        while (!listener.commandComplete) {
            listener.wait();
        }
    }
}
```

The implementation of `sendSequenceAndWait` simply iterates through the array and sends each string as a separate command, after which it waits for the stream of commands to finish:

```
static void sendSequenceAndWait(List<String> sequence)
  throws Exception {
    synchronized(listener) {
        grblController.queueCommands(sequence);
        listener.fileStreamComplete = false;
        grblController.beginStreaming();
        while (!listener.fileStreamComplete) {
            listener.wait();
        }
    }
}
```

All these methods reference a listener that I need to implement to find out when the commands and streams are finished.

```
static class CutterListener
  implements ControllerListener {
```

```
volatile boolean connected;
volatile boolean commandComplete;
volatile boolean fileStreamComplete;
double prbZ;

@Override
public synchronized void fileStreamComplete(
  String string, boolean bln) {
    fileStreamComplete = true;
    notify();
}

@Override
public synchronized void commandComplete(
  GcodeCommand gc) {
    commandComplete = true;
    notify();
}

@Override
public void messageForConsole(
  String msg, Boolean verbose) {
    if (!verbose &&
        msg.startsWith("['$H'|'$X' to unlock]")) {
          synchronized (this) {
            connected = true;
            notify();
        }
    }
    if (!verbose && msg.startsWith("[PRB:")) {
        String pattern =
"\\[PRB\\:-[0-9]*\\.[0-9]*,-[0-9]*\\.[0-9]*," +
"(-[0-9]*\\.[0-9]*)\\:1\\]";
      Matcher matcher =
      Pattern.compile(pattern).matcher(msg);
      if (matcher.find()) {
```

```
            prbZ =
              Double.parseDouble(matcher.group(1));
            }
        }
    }

    // Additional empty methods omitted

}
```

Executing this program homes the router and runs the probe algorithm to test the length of the tool. The last method parses the probe output to find the Z coordinates of the router when the end-mill's tip is touching the probe. To output the probe coordinates and cleanly shut down the router, you can finish by using the following code:

```
System.out.println(
    "PRB_Z = " + listener.prbZ);
TimeUnit.SECONDS.sleep(5);
grblController.closeCommPort();
```

Write down the value of `PRB_Z`, because you will need it for calculating the offset to the workplane.

Calculating the Workspace Location

To figure out the workspace location, you need to manually move the router to touch the surface of the work area. The easiest way to do this is to open the `UniversalGcodeSender` user interface. To run it, use the appropriate platform-specific script or simply type the following at the command line:

```
java -jar UniversalGcodeSender.jar
```

The UI lets you specify the port and the baud rate. Choose the same values you used in your Java program to connect to the CNC router, and then click the **Open** button.

Once the router is connected, it is in a "WARN" state, because the router has not been homed. Navigate to the **Machine Control** tab and click the **$H** button to send a homing command.

Now you can manually control the router head by using the **X**, **Y**, and **Z** movement buttons. Start with a course movement speed to get the head close to the work surface, and then change to a finer movement speed as you get closer. Using a piece of paper, get the head as close to the work surface as you can without touching the paper, as shown in Figure 1.

The value for Z shown under the **Machine Position** section of the screen tells you where the top of your work surface is. Record this value as `WRK_Z`.

To calculate the `PROBE_OFFSET`, use the following formulas:

```
PROBE_OFFSET = PRB_Z -  WRK_Z - 5
```

Plugging in the values that I got from my machine gives me:

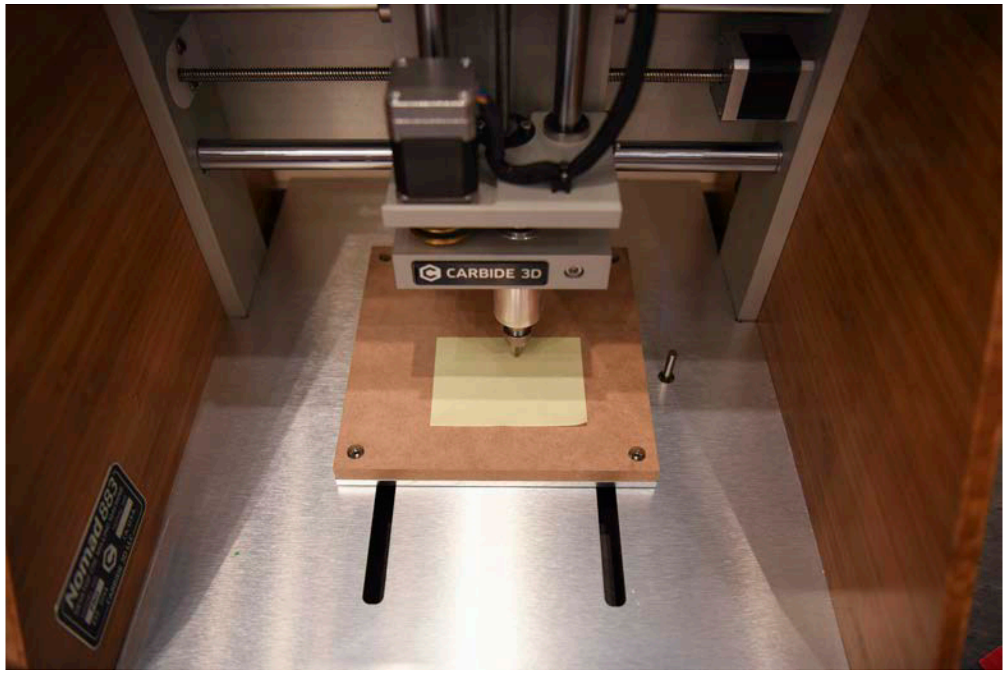


Figure 1. Putting the head close to the work surface

```
PROBE_OFFSET = -85.525 - (-91.57) - 5
PROBE_OFFSET = 1.045
```

Once you have calculated the PROBE_OFFSET for your machine, you can accurately and repeatedly cut through objects to the top of your work surface.

To modify my program to use the new probe offset value I calculated, I am going to reset the coordinate system after performing the homing and probing steps. To do this, I use a G10L20 G-code command.

This G-code command enables you to specify X, Y, and Z values for the new coordinate space. Here is a static string that has the new coordinates specified:

```
static final String COORDINATE_RESET_TEMPLATE
   = "G10 L20 P0 X220 Y205"; // Z is added based on PRB_Z
```

The P0 modifier specifies that you are modifying the default coordinate system. The X and Y values should be about right for any Nomad 883 Pro, but the Z value requires more precision and should be set using the PROBE_OFFSET value you calculated earlier.

To set the Z value, add the following line after the probe sequence:

```
sendCommandAndWait(
    COORDINATE_RESET_TEMPLATE + " Z" +
    (PROBE_OFFSET - listener.prbZ));
```

Building a Path

To demonstrate how you can generate G-code from within Java programmatically, I'll create a simple geometric algorithm to cut out a star pattern. Because I'm doing this in code, I can make the number of points and the size of the inner and outer radius configurable.

The basic algorithm for creating a star is to calculate the location of equally spaced points along a circle. For a circle of radius R and a star with P points, you can find the location of an outer vertex, v, by using the following algorithm:

```
x = cos(v * 360 / P)
y = sin(v * 360 / P)
```

Converting this to radians (where 180 degrees is equal to π radians), you get the following:

```
x = cos(v * 2π / P)
y = sin(v * 2π / P)
```

To complete the algorithm, you also need to calculate the inner vertices, which occur halfway between each of the outer vertices. A more generalized algorithm in Java code that gives you both inner and outer vertices is the following:

```
for (int i=0; i<points * 2; i++) {
    double r = i%2 == 0 ?
        innerRadius : outerRadius;
    double x = Math.cos(i * Math.PI/points) * r;
    double y = Math.sin(i * Math.PI/points) * r;
}
```

Now that I know the vertices for creating a star, I can output G-code that will tell the CNC router to trace the outline. These are the relevant G-code instructions:

- G0: Rapidly move to a specific location.
- G1: Perform a linear move to a location.

Both of these methods take optional X, Y, and Z parameters in decimal format for the location to move to.

To get the starting location of the star, you simply need to calculate the position of the first vertex. Conveniently, sin(0) is 0 and cos(0) is 1, so the earlier algorithm is reduced to the following:

```
x = innerRadius;
y = 0;
```

Using an offset to help center the star in the work area and applying the previously mentioned G-code functions, you get the following function for moving to the starting location of the star:

```
static void moveToStart(
  double innerRadius, double offset)
  throws Exception {
    sendCommandAndWait(
        "G0X" +
        String.format("%.3f", innerRadius + offset) +
        "Y" +
        String.format("%.3f", offset) +
        "Z" +
        String.format("%.3f", MATERIAL_THICKNESS + 1));
}
```

This uses a modified version of the sendSequenceAndWait function that takes a single command. Here is the code for that function:

```
static void sendCommandAndWait(String sequence)
  throws Exception {
    synchronized(listener) {
        grblController.queueCommand(sequence);
        listener.fileStreamComplete = false;
        grblController.beginStreaming();
        while (!listener.fileStreamComplete) {
            listener.wait();
        }
    }
}
```

And the full drawStar function to create the G-code for tracing all of the points of the star is shown here:

```
static List<String> drawStar(
  int points, double innerRadius,
  double outerRadius, double offset) {
    List<String> gcode =
        new ArrayList<>(points * 2 + 1);
    for (int i=0; i<points * 2; i++) {
        double r = i%2 == 0 ?
            innerRadius : outerRadius;
        double x =
            Math.cos(i * Math.PI/points) * r;
        double y =
            Math.sin(i * Math.PI/points) * r;
        gcode.add("G1X" +
            String.format("%.3f", x + offset) +
            "Y" +
            String.format("%.3f", y + offset));
    }
    gcode.add(gcode.get(0));
    return gcode;
}
```

To complete the example, I need a few extra G-code sequences:

```
static final List<String> START_SPINDLE =
     Arrays.asList("G21", "G90", "M3 S9000");
static final List<String> END_SEQUENCE =
     Arrays.asList("M5", "$H", "M30");
```

These sequences are used at different times. The first sequence in the code above sets the units to metric (G21), sets absolute distance mode (G90), and starts the spindle (M3 S9000). The second sequence is an end sequence that stops the spindle (M5), homes the head ($H), and then

ends the program (M30).

I also need to support multiple passes through the material, because acrylic is too hard to cut in one pass. For a 1/8-inch thick piece of acrylic, I recommend seven passes, each incrementally deeper. The following code calls the drawStar function multiple times to accomplish that:

```
sendSequenceAndWait(START_SPINDLE);
moveToStart(50, 100);
for (int i = 1; i <= Z_STEPS; i++) {
    double newZ = MATERIAL_THICKNESS *
        (Z_STEPS - i) / Z_STEPS;
    sendCommandAndWait(
        "G1Z" + String.format("%.3f", newZ) +
         "F355.600");
    sendCommandAndWait("F1117.600");
    sendSequenceAndWait(
         drawStar(9, 50, 90, 100));
}
sendSequenceAndWait(END_SEQUENCE);
```

This code makes use of a new argument, F, to G1. It is the feed rate. You can also call this code by itself to set the feed rate for subsequent linear operations. Finally, there are two parameters, the MATERIAL_THICKNESS and number of Z_STEPS, that you can set based on the material you are using:

```
static final int Z_STEPS = 7;
static final double
   MATERIAL_THICKNESS = 25.4 * 1/8;
```

Just to make sure the program is working correctly, you might want to perform a first run with no material inside the machine. If anything unexpected happens, killing the Java program or pressing the power switch on the Nomad 883 Pro will immediately stop the operation.

Once you are confident that you have a working application, load a piece of acrylic plastic into your Nomad 883 Pro, and run the program again. The router should slowly cut the acrylic layer by layer until it gets just above the router's wasteboard.

Upon completion, vacuum up the plastic scraps and carefully remove the finished piece from the tray. You should have a perfectly cut star similar to the one shown in **Figure 2**.

Running the Code on the Raspberry Pi

Running the code on the Raspberry Pi is the easiest part of the process described in this article. Java 8 comes bundled with the standard Raspbian Linux distribution, so if you have an up-to-date installation, you already have Java ready to go. And if you are using NetBeans, there is built-in support for running the code on the Raspberry Pi using the Remote Java Platform functionality.

To set up your Raspberry Pi in NetBeans for the first time:

1. In the **Tools** menu, choose **Java Platforms**.
2. Click **Add Platform** and select **Remote Java Standard Edition**.

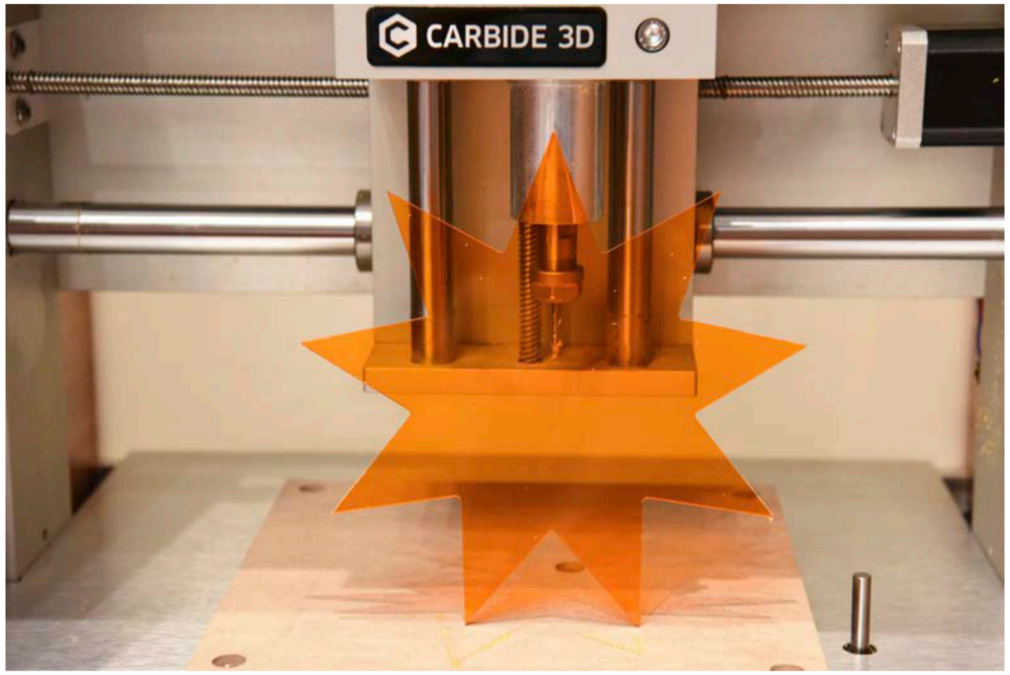


Figure 2. An acrylic star cut using the presented code

3. Name your platform whatever you like and set the following values (as shown in Figure 3):
   a. For **Host**, specify your Raspberry Pi's IP address.
   b. For **Username**, specify the SSH user (the default is `pi`).
   c. For **Password**, specify the SSH password (the default is `raspberry`).
   d. For **Remote JRE Path**, specify the location of Java 8. Note: Use `sudo update-alternatives --display java` to find the path.
4. Click **Finish**.

Now you have a working JRE setup. To enable this in your project, open the Project Properties dialog box, select the **Run** category, and from the **Runtime Platform** list, select the new platform you created. You will be prompted to save the new configuration, which you can name anything you like.

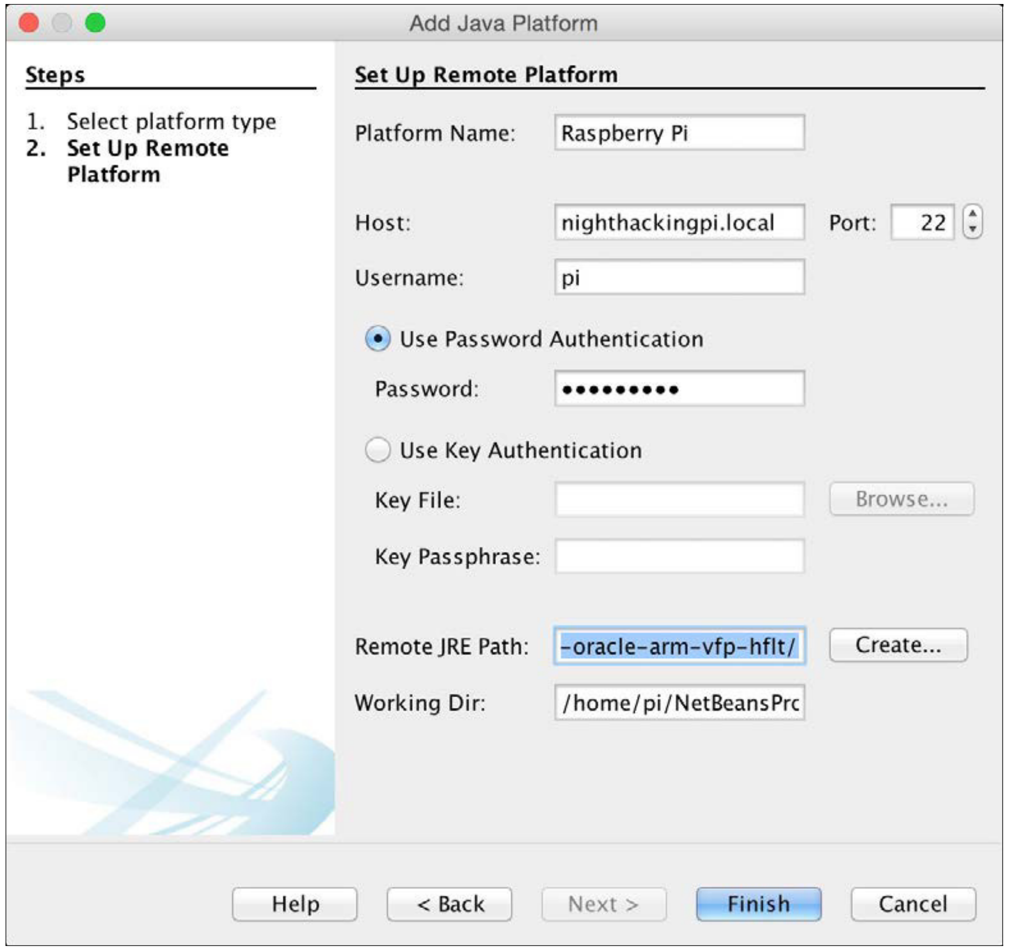


Figure 3. The configuration screen in NetBeans

The final step is to modify the code to change the port name to match how the Nomad 883 Pro shows up under Raspbian Linux. This port name will be available when you run the `dmesg` command after connecting the router via a USB. For me, the following value worked:

```
static String PORT_NAME = "/dev/ttyACM0";
```

Once this is all set up, you can run, debug, and profile the sample code I provided on the Raspberry Pi right from your IDE. The advantage of running the code on the Raspberry Pi is that you can execute the application without having a computer hooked up, which frees your expensive laptop or desktop from issuing repetitive serial commands. Also, because the Raspberry Pi is a dedicated device, the chance of timing delays or system crashes is greatly reduced.

For more information about running Java on the Raspberry Pi, check out the book *Raspberry Pi with Java: Programming the Internet of Things (IoT)* from Oracle Press. [A chapter from that book appeared in the May/June 2015 issue of *Java Magazine*. —*Ed.*]

Conclusion

What this project shows is that a lot of the basic programming of devices consists of configuring the setup to receive commands from a Java program and then sending those commands to the device in a format it understands. The Java tooling and its large ecosystem of software for devices makes this particularly easy. </article>

Stephen Chin is the lead Java community manager at Oracle Technology Network, author of *Raspberry Pi with Java*, coauthor of *Pro JavaFX Platform*, and JavaOne community chair. He is a five-time Rock Star Award recipient. Chin interviews hackers in their natural habitat and posts the videos on http://nighthacking.com/.

ERIC BRUNO

Using the Cloud with IoT

By pushing device control and analytics into an IoT-specific cloud, devices can be remotely managed and operated.

The growing area of the Internet of Things (IoT) is an end-to-end discipline involving the integration of remote devices (sensors, controllers, actuators, and so on) with server-side analytics, monitoring, database storage, and enterprise integration. Reliable distributed communication is also a must. Because of the many platforms, devices, and discrete activities that must be bound together, the multi-platform nature of Java makes it an ideal language to work in.

What's frequently needed, however, is a platform or framework to help organize all of the related business and technical activities that go into an IoT application. This is where the cloud can help.

With respect to the cloud, there are three important points to remember:

- The cloud is more than someone else's data center. Look past infrastructure and toward platform as a service (PaaS), where you're insulated from details such as servers, load balancers, clusters, database sizes, and configuration. Instead, focus on solving problems and writing code.
- PaaS services offer ways to implement analytics that don't replace the fun of Java coding, but instead reduce tedious tasks.
- The cloud puts the power of customer- and business-driven changes in the hands of business owners. This reduces the need for developers to make minor changes and tweaks.

For instance, by offloading key analytics from within Java to the cloud, you leave the task of tuning threshold values, monitoring details, and dealing with other tunable parameters to business owners who know the problem domain and are in touch with users. In this article, I explore this approach with an example using Oracle Internet of Things Cloud Service (hereafter called Oracle IoT Cloud Service or IoTCS).

Using Java and Oracle IoT Cloud Service

Being a PaaS service, Oracle IoT Cloud Service enables you to focus on building solutions instead of worrying about servers, software updates, manual deployment, load balancing, and other infrastructure issues.

Rather than offering just plain-vanilla PaaS services, providers increasingly offer specialized implementations and frameworks in the cloud, such as Oracle IoT Cloud Service, to make it easier for you to build applications with less effort, management, and code.

For IoT analytics, this often means being able to modify thresholds, add new alerts based on new conditions, and remove old alerts. Oracle IoT Cloud Service makes these tasks simple with its built-in Streams Explorer analytics rules. To see this in action, I'll examine what it takes to build a cloud application using these services.

Looking at the IoTCS documentation, you'll notice many REST APIs, rules-based analytics, and integration with business intelligence, mobile devices, and enterprise systems based on packages and adapters. It might not be apparent how Java plays a role. However, Oracle provides a Java-based IoTCS Client Software Library that facilitates device, gateway, and enterprise application development in Java.

For example, IoTCS uses OAuth 2.0 tokening to implement much of its security model. Using the REST APIs, your device would need to read an SSL keystore, Base64-encode the appropriate username and password, send the proper OAuth 2 handshake messages, and handle the responses—all to activate an IoT device and have it communicate data. Doing this with pure REST can be tedious. However, by using parts of the Oracle IoT Cloud Service Java Client Software Library, the code to authenticate and activate an IoT device is reduced to what's shown in **Listing 1**.

Oracle provides a Java-based Oracle IoT Cloud Service Client Software Library that facilitates device, gateway, and enterprise application development in Java.

Listing 1.

```
String url = ...
HttpClient httpClient =
    new HttpClient(server_url,uid,pwd);

WebResource resource =
    httpClient.createHTTPClient(true).resource(url);

TokenInputDetails tokenInputDetails =
    new TokenInputDetails();

tokenInputDetails.setTokenType(
    TokenType.TOKEN_TYPE_ACTIVATION);
tokenInputDetails.setDeviceId(endpointID);
tokenInputDetails.setSharedSecret(secret);

String deviceModelStr =
    "urn:com:acme:conveyorbeltmodel";
String messageFormat =
    "urn:com:acme:conveyorbeltmodel:speed";

Authorization authorization =
    new Authorization(resource);
PrivateKey privateKey =
    authorization.activate(tokenInputDetails);
```

In fact, this code can be reduced further to the following using the `oracle.iot.client.device.DirectlyConnectedDevice` class from the IoTCS Client Software Library (see **Listing 2**).

Listing 2.

```
DirectlyConnectedDevice device =
    new DirectlyConnectedDevice();
if ( ! device.isActivated() ) {
    device.activate( getDeviceModelURN() );
}
```

When leveraging the cloud-based development paradigm, the key is understanding which aspects of an IoT solution to leave as cloud-handled analytics and which to implement in code. The goal is to alleviate tedious maintenance requirements by allowing the business to make changes without new software deployments. To demonstrate, I describe an IoT application scenario, dive into the Java code, and explore the Oracle IoT Cloud Service setup that makes this goal achievable.

Sample Application: Monitoring Industrial Conveyor Belts

In this sample IoT application, a factory conveyor belt is implemented, monitored, and controlled using Java and IoTCS. Although the belt motor is emulated here, the Java code is written with the IoTCS Client Software Library to run on an actual device or on a gateway connecting the device to the cloud. It's assumed that the conveyor belt in this scenario has the ability to connect to the internet directly to send

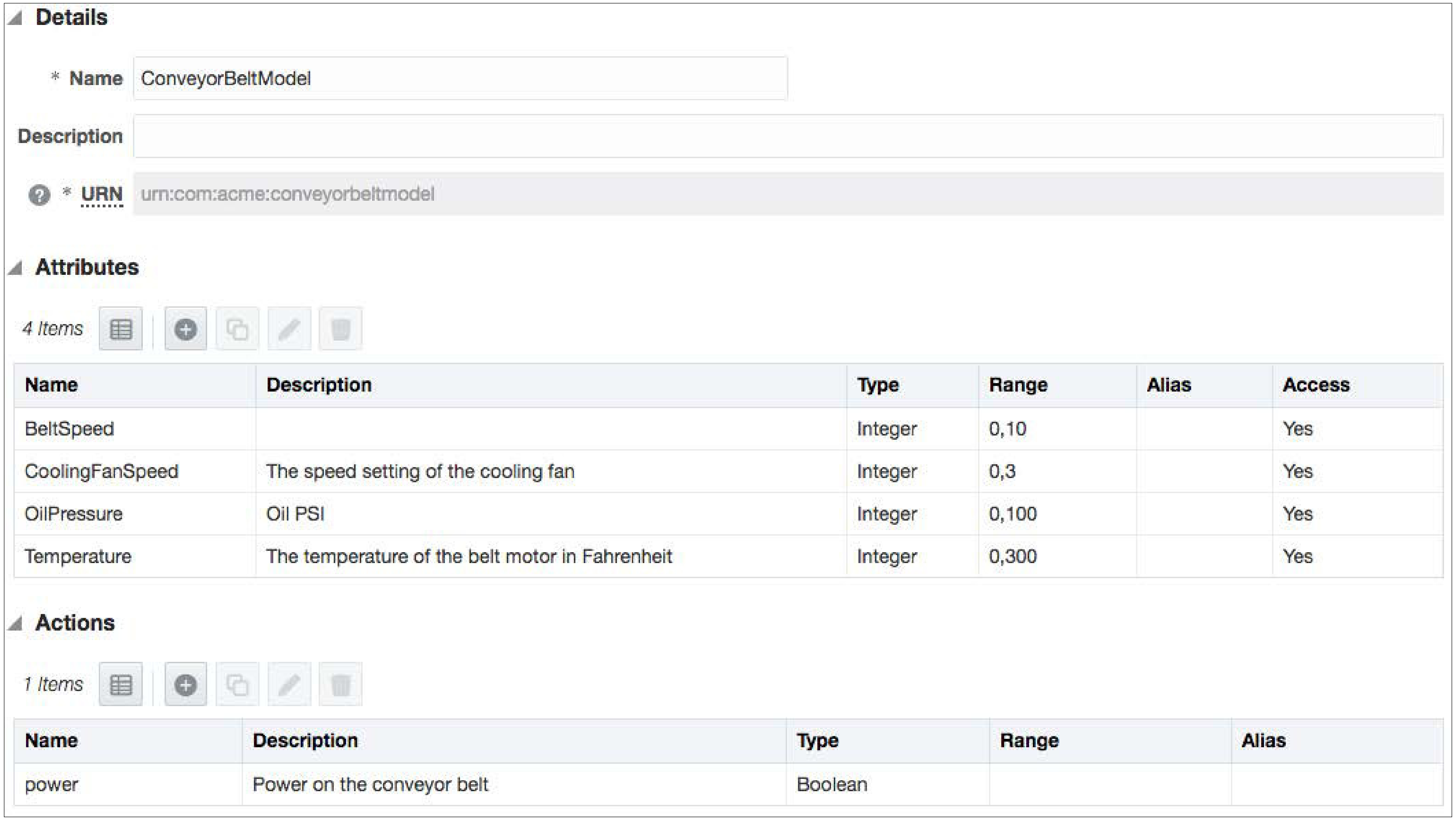
attribute updates and listen for actions.

The first step is to create a new *device model* in the IoTCS instance, as shown in **Figure 1**. Here, the device's data attributes and actions are defined.

In this image, device attributes indicate belt speed, cooling fan speed, oil pressure, and temperature, and an action is created to power the belt on or off.

On the server side, you define an IoTCS application to maintain the device data analytics and integration with other cloud services or applications. The illustration in **Figure 2** shows the overall architecture of the sample end-to-end IoTCS application. In this case, the monitoring application is built in Java and runs on premises—on my laptop—but it can also run in the cloud.

Analytics are defined to handle the threshold processing to indicate when the cooling fan needs to be turned on or the belt needs to be shut down (for example, due to overheating or low oil pressure), and to handle the integration with a Java monitoring application. **Figure 3** shows the implementation of the TempThreshold1 analytic, defined using Boolean logic



Details

* Name: ConveyorBeltModel

Description:

* URN: urn:com:acme:conveyorbeltmodel

Attributes

4 Items

Name	Description	Type	Range	Alias	Access
BeltSpeed		Integer	0,10		Yes
CoolingFanSpeed	The speed setting of the cooling fan	Integer	0,3		Yes
OilPressure	Oil PSI	Integer	0,100		Yes
Temperature	The temperature of the belt motor in Fahrenheit	Integer	0,300		Yes

Actions

1 Items

Name	Description	Type	Range	Alias
power	Power on the conveyor belt	Boolean		

Figure 1. The conveyor belt's device model in the Oracle IoTCS administrative console

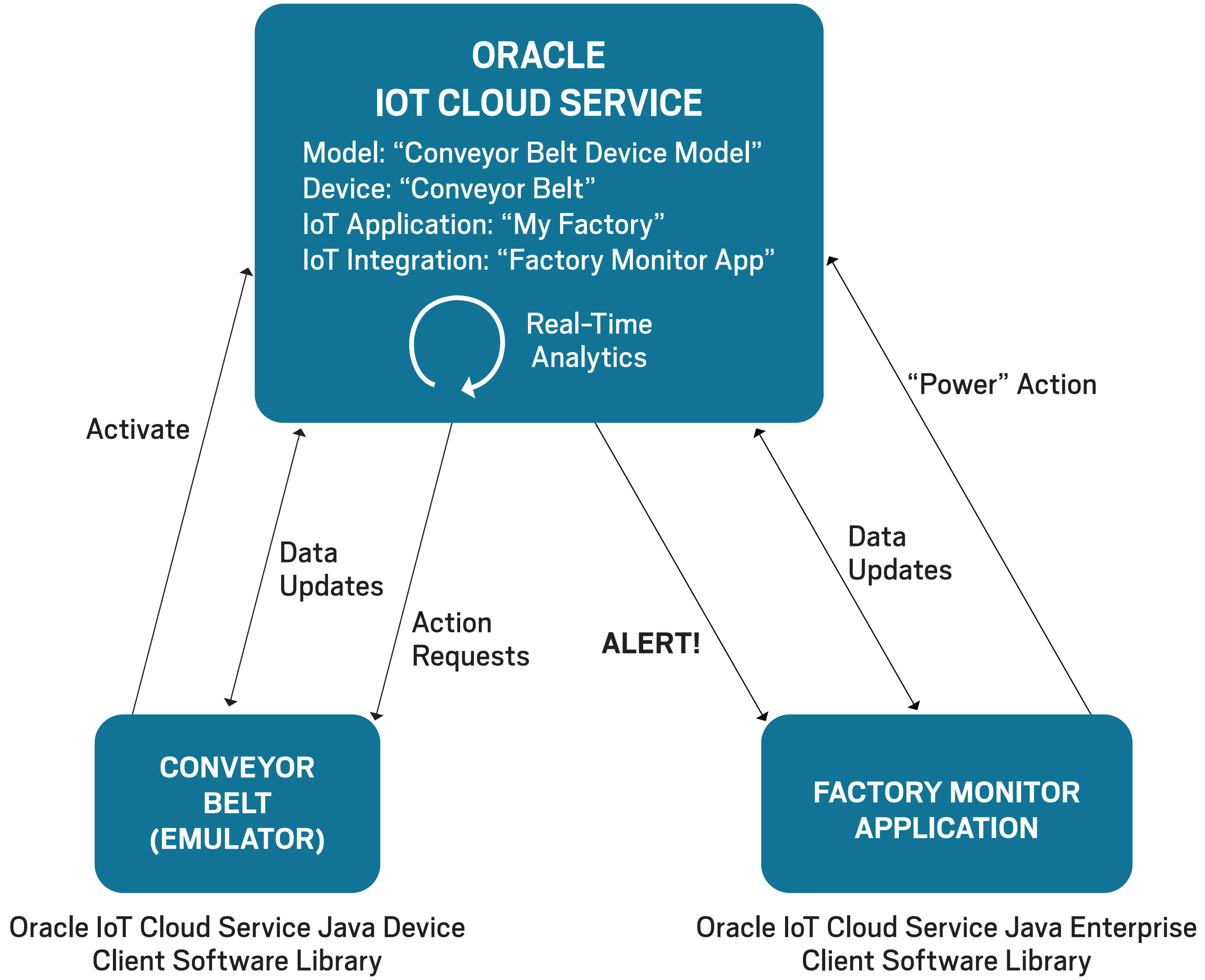


Figure 2. The architecture of the sample Oracle IoTCS application

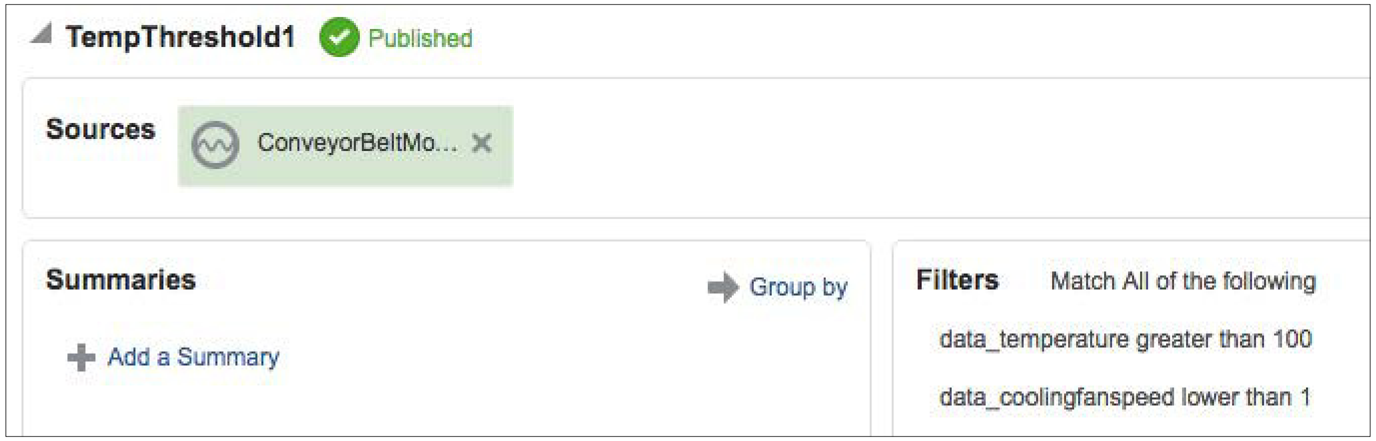


Figure 3. A Streams Explorer analytic definition

with the appropriate device attributes.

The data source is the conveyor belt motor, and the logic should send an alert when the motor temperature goes above 100 degrees Fahrenheit and the cooling fan speed is less than 1 (the fan is not on). The TempThreshold2 analytic is similar, except that it looks for a higher temperature (> 150 degrees Fahrenheit) and a fan speed that is less than 2. The remaining analytics are similar.

However, these analytics are useless unless there's an application that listens for the resulting alerts to act on them. This is the Java monitoring application mentioned previously, which is integrated through a REST endpoint to which IoTCS sends the analytics-driven alerts. As a result, the monitoring application instructs the device to change its belt motor speed, adjust the cooling fan speed, or shut down in response to the analytics alerts (see **Listing 3**).

Listing 3.

```
public static void handleREST(HttpExchange t)
  throws Exception {
    // Read request JSON string
    BufferedReader bodyReader =
        new BufferedReader(
            new InputStreamReader(
                    t.getRequestBody() ));
    //...

    // Parse the JSON string
    StringReader sr =
        new StringReader(alert.toString());
    JsonReader jr = Json.createReader(sr);
    JsonObject jo = //...
    JsonObject payload =
        jo.getJsonObject("payload");

                         // Get the alert type by URN
```

```
    String alertType =
        payload.getString("format");

    switch ( alertType ) {
        case TEMP_TOO_LOW_ALERT_URN:
            device.set(FAN_SPEED_ATTR,
                       OFF);
            break;
        case TEMP_LOW_ALERT_URN:
            device.set(FAN_SPEED_ATTR,
                       FAN_LOW_SPEED);
            break;
        case TEMP_MED_ALERT_URN:
            device.set(FAN_SPEED_ATTR,
                       FAN_MED_SPEED);
            break;
        case TEMP_HIGH_ALERT_URN:
            device.set(FAN_SPEED_ATTR,
                       FAN_HIGH_SPEED);
            break;
        case TEMP_CRITICAL_ALERT_URN:
            shutDownBelt(device);
            break;
        //...
    }
}
```

With this IoTCS analytics-based implementation, changing the thresholds that indicate when the cooling fan speed needs to be increased or decreased doesn't require changes to the Java application. The analytics can be set and adjusted instead by someone who understands the mechanical properties of the belt motor. The alternative would be the much less satisfactory hard-coding of values and limits, using code such as shown in Listing 4.

Listing 4.

```
VirtualDevice device = event.getVirtualDevice();
int temp = (Integer)namedValue.getValue();
int coolingFanSpeed = device.get(FAN_SPEED_ATTR);

// Determine if action is needed based on new temp
if ( temp >= 300 ) {
    // Shut things down!
    device.set(BELT_SPEED_ATTR, 0);
}
else if ( temp > 250 && ...
```

To receive notification that the monitoring application changed the belt or fan speed, the device needs to register for appropriate attribute changes, as shown in Listing 5.

Listing 5.

```
DeviceModel model =
    device.getDeviceModel(
        CONVEYOR_BELT_MODEL_URN);

VirtualDevice virtualDevice =
    device.createVirtualDevice(
        device.getEndpointId(),
        model);

virtualDevice.setOnChange(
   BELT_SPEED_ATTR,
   new VirtualDevice
       .ChangeCallback<VirtualDevice>() {
          public void onChange(
           VirtualDevice
             .ChangeEvent<VirtualDevice> e) {
                  onBeltSpeedChange(e);
           }
});
```

```
virtualDevice.setOnChange(
   COOLING_FAN_SPEED_ATTR,
   new VirtualDevice.ChangeCallback<VirtualDevice>() {
       public void onChange(
         VirtualDevice.ChangeEvent<VirtualDevice> e) {
             onFanSpeedChange(e);
         }
});
```

For cleaner code, I use a Java 8 lambda expression, as shown in **Listing 6**. The callback method—not shown—simply switches on the attribute name to react to the value change accordingly.

Listing 6.

```
virtualDevice.setOnChange(
    (VirtualDevice.ChangeEvent<VirtualDevice>
        event) -> {
    onAttributeChange(event);
});
```

To listen for IoTCS action requests (such as the power action), the device needs to implement a separate callback (see **Listing 7**). The callback method needs to define the parameters to match those defined in the IoTCS device model.

Listing 7.

```
virtualDevice.setCallable("power",
   new VirtualDevice.Callable<Boolean>() {
   public void call(
     VirtualDevice virtualDevice, Boolean on) {
       belt.power(on);
     }
});
```

Of course, devices can update their own attributes. To enable them to do so, use the IoTCS Client Software Library to ensure that changes are sent to the cloud efficiently. You do this through the concept of a virtual device, which is the cloud's representation of the device itself. The code in **Listing 8** starts the process with a call to `VirtualDevice.update`. Next, each attribute value is assigned with a call to `VirtualDevice.set`. Finally, a call to `VirtualDevice.finish` indicates the changes can be sent to IoTCS. Under the covers, the client library uses reliable REST-based messaging to send changes to IoTCS.

Listing 8.

```
virtualDevice.update()
    .set(BELT_SPEED_ATTR, beltSpeed)
    .set(TEMP_ATTR, temp)
    .set(FAN_SPEED_ATTR, fanSpeed)
    .set(OIL_PRESSURE_ATTR, oilPSI)
    .finish();
```

To receive device attribute updates, the monitoring application registers just as the device does. The code in **Listing 9** shows this as well as how to call actions and change an attribute value on a device, all from the monitoring application.

Listing 9.

```
EnterpriseClient ec =
    EnterpriseClient.newClient(APP_NAME);

VirtualDevice virtualDevice =
    ec.createVirtualDevice(
        "0-LENQ", deviceModel);

virtualDevice.setOnChange( /*...*/ );
virtualDevice("power", true);
virtualDevice.set(BELT_SPEED_ATTR, 1);
```

In this example, the monitoring application turns the conveyor belt on by calling the `power` action with the right parameter value and then setting the belt speed.

At this point, everything is in place for the end-to-end conveyor belt IoT application to function properly. In reality, the application would have a user interface to more easily view the activity, but for my purposes, the command line will do fine. The following listing shows some sample output from the conveyor belt device emulator as it receives commands from the monitoring application. [Indented lines are wrapped from the previous lines. —*Ed.*]

```
Created the directly-connected device
--Device model=urn:com:acme:conveyorbeltmodel
--Instance=0-LENQ
--Starting belt speed:0
Motor temp: 70
Motor oil pressure: 0
Mon Jul 25 12:00:37 EDT 2016 :
   0-LENQ : Call : "power"=true
Motor temp: 70
Motor oil pressure: 0
Mon Jul 25 12:00:50 EDT 2016 :
   0-LENQ : onChange : BeltSpeed was: 0, now: 1
Motor temp: 81
Motor oil pressure: 40
Motor temp: 82
Motor oil pressure: 42
...
Mon Jul 25 12:07:50 EDT 2016 :
  0-LENQ : onChange : CoolingFanSpeed was: 0, now: 1
...
```

IoTCS analytics alert the monitoring application regarding when to turn the cooling fan on or off, as well as about other important events such as low oil pressure and overheating events. The following output shows the monitoring application reacting to the various IoTCS analytics alerts.

```
Listening on http://localhost:7890/monitor/messages
Device: 0-LENQ, OilPressure: 49
Device: 0-LENQ, Temperature: 71
Alert: 0-LENQ, TempThreshold1 - set cooling fan to 1
...
```

What should be noted from this output, and the corresponding code shown earlier, is how the device and the monitoring application both react to events from IoTCS, as implemented through the Java Client Software Library and driven by analytics defined in IoTCS itself. The cloud abstracts much of the tedious configuration and glue to make this application functional.

Conclusion

As you can see, a PaaS cloud restores some of the fun to coding. It removes a level of monotony, and it enables users to have greater agility for adjusting applications to their needs. </article>

Eric Bruno is a principal sales consultant and an Oracle IoT Champion in the Oracle ISV software sales team. He has more than 20 years' experience in the information technology community as an enterprise architect, developer, and industry analyst with expertise in full lifecycle, large-scale software architecture, design, and development.

learn more

[Oracle IoT Cloud Service home page]

[OAuth 2.0]

SEBASTIAN DASCHNER

JAX-RS.next: A First Glance

A look at what's coming next in JAX-RS 2.1

JAX-RS is the standard for developing RESTful HTTP web services in an enterprise application. It offers a productive yet powerful way of defining REST resources by annotating plain old Java objects (POJOs) that contain the business logic without needing to write the HTTP plumbing by hand. Due to the fact that RESTful web services—or HTTP services implementing some of the REST constraints—are commonly used in enterprise projects and also because of its productive programming model, the JAX-RS standard is widely adopted among Java developers.

Java EE 7 comes with JAX-RS 2.0 (JSR 339), whereas Java EE 8, which is currently in progress, will contain JAX-RS 2.1 (JSR 370).

This article explains some of the new concepts and features the specification's update will deliver. These include support for server-sent events (SSEs), better integration with other Java EE specifications, and integration of reactive programming functionality, as well as non-blocking I/O on the client side. To follow along, you need some experience with JAX-RS.

Note: The specification and all the topics and examples this article covers are still subject to change. This article, then, is a preliminary explanation of what very likely will be included in the next JAX-RS version.

Server-Sent Events

SSEs are a new technology that is part of HTML5. SSEs automatically receive updates from a server via HTTP, and they offer an easy-to-use, one-way streaming communication protocol that perfectly fits use cases that have broadcast updates—for example, social media updates, stock prices, or news feeds.

The server pushes UTF-8 text-based data as content type `text/event-stream` to a client that previously connected to the streaming endpoint and, therefore, registered for the events. The format of the events looks as follows:

```
data: This is a message

event: namedmessage
data: This message has an event name

id: 10
data: This message has an id which will be sent as
 'last event ID' if the client reconnects
```

This approach of asynchronous events over an established connection competes with the more powerful WebSockets standard, which supports bidirectional communication. The main advantages of SSEs, however, are that they are compliant with HTTP technology, because they use HTTP as the communication protocol (which makes it easy to integrate them into existing networks or developer tools), and they natively support event IDs and reconnects.

JAX-RS 2.1 will support SSEs for both JAX-RS resources and the clients.

The server side of SSEs. The following code shows an SSE-enabled JAX-RS endpoint.

```
@Path("events-examples")
@Singleton
public class EventsResource {

    @Inject
    SseContext sseContext;

    private SseBroadcaster sseBroadcaster;
    private int lastEventId;
    private List<String> messages = new ArrayList<>();

    // see methods initSse(), itemEvents(), newMessage()
    // and createEvent() below
}
```

The EventsResource class is a singleton Enterprise JavaBean (EJB). It contains the SseBroadcaster, which is connected to all clients. SseContext is an injectable context object responsible for creating SSE broadcasters, outputs, and events. The messages represent the history of all created messages. The sseBroadcaster is initialized and enhanced with optional SseBroadcaster.Listeners in the @PostConstruct method.

```
@PostConstruct
public void initSse() {
    sseBroadcaster = sseContext.newBroadcaster();

    sseBroadcaster.register(
     new SseBroadcaster.Listener() {
        @Override
        public void onException(SseEventOutput output,
                Exception exception) {
            // ...
        }

        @Override
        public void onClose(SseEventOutput output) {
            // ...
        }
    });
}
```

The clients connect against the following JAX-RS resource:

```
@GET
@Produces(MediaType.SERVER_SENT_EVENTS)
@Lock(LockType.READ)
public SseEventOutput itemEvents(
        @HeaderParam(HttpHeaders.LAST_EVENT_ID_HEADER)
        @DefaultValue("-1") int lastEventId) {
    final SseEventOutput eventOutput =
            sseContext.newOutput();

    if (!sseBroadcaster.register(eventOutput)) {
        // try to make the client reconnect after
        //     5 seconds
        throw new ServiceUnavailableException(5L);
    }

    if (lastEventId >= 0) {
        // replay messages
        try {
            for (int i = lastEventId;
                    i < messages.size(); i++) {
                eventOutput.write(createEvent(
                        messages.get(i), i + 1));
            }
        } catch (IOException e) {
            throw new InternalServerErrorException(
                "Could not replay messages ", e);
        }
    }
```

```
    return eventOutput;
}

private OutboundSseEvent createEvent(String message,
        int id) {
    return sseContext.newEvent()
            .id(String.valueOf(id))
            .data(message)
            .build();
}
```

The `SseEventOutput` return type tells the JAX-RS implementation to keep the client connection open and capable of sending events through the broadcaster. Content type `text/event-stream` is used for SSEs. The SSE standard specifies that the `Last-Event-ID` header controls the event stream of the previously received events; that is, if SSE is supported by the server, the server resends the events created after `lastEventId`.

The method registers the newly created output to the `sseBroadcaster` and then immediately resends the events if the corresponding HTTP header has been sent. After the output is registered to the broadcaster, the client—together with all other active clients—receives events, which are created using the `SseContext` injected to the resource class.

This process is shown in the following code:

```
@Schedule(second = "*/5", minute = "*", hour = "*",
        persistent = false)
@Lock(LockType.WRITE)
public void newMessage() {
    final String message =
        "It's now: " + LocalDateTime.now();
    messages.add(message);

    final OutboundSseEvent event = createEvent(
        message, ++lastEventId);

    sseBroadcaster.broadcast(event);
}
```

Here, the `@Schedule` method is called every five seconds to create new events and broadcast them to all connected clients. I store the messages for later reuse in case any reconnecting client asks to continue the event stream from the last received message.

The client side of SSEs. JAX-RS 2.1 will also include client-side functionality to consume SSEs. There are two ways of creating a connection to an endpoint.

`SseEventSource` offers functionality to open a connection to an SSE endpoint by registering an event listener and, thus, providing a reactive way of handling the events.

```
public class SseClient {

    private final WebTarget target =
            ClientBuilder.newClient().target("...");
    private SseEventSource eventSource;

    public void connect(Consumer<String> dataConsumer){
        eventSource = SseEventSource.target(target)
            .register(ev ->
               dataConsumer.accept(ev.readData()))
            .open();
    }

    public void disconnect() {
        if (eventSource != null)
            eventSource.close();
    }
}
```

The SseEventSource is created by calling the target method on a WebTarget, registering an SseEventSource.Listener, and opening the connection. After successfully opening the connection, the current thread continues and the listener—which in this case would be dataConsumer .accept—is called as soon as events arrive.

Consumers can call start() and stop() **to resume and pause the stream,** and all events that happen during the pause are replayed after resuming.

SseEventSource handles all required plumbing, including reconnecting after a connection loss, by sending an appropriate Last-Event-ID header and then handling Retry-After headers sent from the server appropriately.

If a more sophisticated way is needed, for instance, to manually control the Last-Event-ID header, you can manually request an SseEventInput from the server.

```
public class StatefulSseClient {

    private final WebTarget target =
            ClientBuilder.newClient().target("...");
    private final Consumer<String> dataConsumer;
    private String lastEventId;
    private SseEventInput eventInput;

    public StatefulSseClient(
            Consumer<String> dataConsumer) {
        this.dataConsumer = dataConsumer;
    }

    public void start() {
        eventInput = target
          .request(MediaType.SERVER_SENT_EVENTS)
          .header(HttpHeaders.LAST_EVENT_ID_HEADER,
                  lastEventId)
          .get(SseEventInput.class);

        new Thread(() -> {
            while (!eventInput.isClosed()) {
                final InboundSseEvent event =
                        eventInput.read();
                if (event != null) {
                    lastEventId = event.getId();
                    dataConsumer.accept(
                        event.readData());
                }
            }
        }).start();
    }

    public void stop() {
        if (eventInput != null &&
            !eventInput.isClosed())
            try {
                eventInput.close();
            } catch (IOException e) {
                // suppress
            }
    }
}
```

By requesting the input directly, all required information has to be sent manually and potential reconnects need to be handled manually. Therefore, the client sends the text/event-stream HTTP header and expects the response to be an SseEventInput type that is handled appropriately by the JAX-RS implementation. This event input is used to receive the actual incoming events.

Because the eventInput.read method blocks until some

event is being sent, you need to manually take care of threading. Therefore, you start a new thread that waits for the events and, thus, the execution in the start method can continue.

Consumers of the StatefulSseClient class can call start() and stop() to resume and pause the stream, and all events that happen during the pause are replayed after resuming.

Reactive Programming

On the client side, another interesting new feature will be the integration of reactive computation types, such as CompletionStage, into the JAX-RS client.

As an example, I'll call several web services and combine the results with functionality that is available in JAX-RS 2.0 already.

Suppose I have a slow HTTP endpoint returning a randomly created response after a while, and suppose I want to consume this service several times and calculate a result from all responses. Polling the services sequentially and then constructing the result would lead to long wait times while the threads are blocked. To avoid that, I can use constructs such as CompletableFutures together with Java 8 lambdas and streams.

Here's a JAX-RS 2.0 client-side example:

```
final Executor executor =
    Executors.newCachedThreadPool();

// endpoint takes 2 seconds to respond
//    with data like {"random":1}

final OptionalDouble average =
   IntStream
     .range(0, 10).parallel()
     .map(i ->
        CompletableFuture.supplyAsync(() ->
          target
            .request(MediaType.APPLICATION_JSON_TYPE)
            .get(JsonObject.class), executor)
            .thenApply(o -> o.getInt("random"))
            .join()
        ).average();

System.out.println(
    "average = " + average.getAsDouble());
```

Calling the endpoint via the target results in a 2-second wait time. To avoid this, I wrap the action in a CompletableFuture and extract the total value from the JSON data once the result is available. The executor is provided as a parameter so you can manage the threads for the 10 concurrent polls while most of the calling threads are waiting. The execution blocks when join() is called, but because I'm running the IntStream in parallel, all the results are available at roughly the same time—after about 2 seconds, rather than after 20 seconds, as they would be when polling one after the other. (The stream of results is then aggregated into the average value.)

This is what is possible today using JAX-RS 2.0 and CompletableFuture to wrap long-running jobs.

JAX-RS 2.1 will integrate reactive invokers for the client side. Constructing invocations is enhanced with rx() methods that wrap responses into a reactive invocation type and, optionally, use a specific ExecutorService. Here's an example:

```
final ExecutorService executor =
        Executors.newCachedThreadPool();

final OptionalDouble average = IntStream
        .range(0, 10).parallel()
        .map(i -> target
           .request(MediaType.APPLICATION_JSON_TYPE)
```

```
            .rx(executor)
            .get(JsonObject.class)
            .thenApply(o -> o.getInt("total"))
            .toCompletableFuture().join()
        ).average();

System.out.println("average = " +
        average.getAsDouble());
```

Calling the `rx()` method configures the builder to use an `RxInvoker`, and the `CompletionStage` is used as the default invocation type. Therefore, all subsequent actions, such as `get()`, return the responses wrapped in this corresponding reactive type. As in the previous example, I chain several actions together and `CompletionStage.toCompletableFuture` enables me to call `join()` as well.

This is one of the many useful improvements in the next release of JAX-RS.

Non-Blocking I/O

Another extension to both the client and the server side will be support for non-blocking I/O (NIO). By using NIO, the caller is guaranteed to be able to call certain methods for either sending data without being blocked or registering a reader that will eventually be called when data is available.

Server-side NIO. The JAX-RS server-side programming model will be enhanced so that developers can register `NioReaderHandler`s for responses or `NioWriterHandler`s for requests, respectively. These callback handlers are called when data needs to be read or written without blocking.

The following examples show how to read and write data in a non-blocking way in a JAX-RS resource:

```
@GET
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response download() {
    final InputStream in = // ...
    final byte[] buffer = new byte[1000];

    return Response.ok().entity(
      out -> {
             try {
                 final int length = in.read(buffer);
                 if (length >= 0) {
                     out.write(buffer, 0, length);
                     return true;
                 }
                 in.close();
                 return false;
             } catch (IOException e) {
                 throw new WebApplicationException(e);
             }
            }).build();
}
```

The `Response.ok().entity` method takes an `NioWriterHandler` as an argument and, thus, the lambda implementing the `NioWriterHandler.write(NioOutputStream)` method is called each time `out` is ready to accept data. The handler method is expected to return a boolean value indicating whether there is more data to write.

A similar handler approach is realized for reading uploaded data from the client's request:

```
@POST
@Consumes(MediaType.APPLICATION_OCTET_STREAM)
public void upload(@Context Request request) {
    final ByteArrayOutputStream out = // ...
    final byte[] buffer = new byte[1000];

    request.entity(
        in -> {
```

```
            try {
                if (in.isFinished()) {
                    out.close();
                } else {
                    final int length =
                        in.read(buffer);
                    out.write(
                        buffer, 0, length);
                }
            } catch (IOException e) {
                throw new
                   WebApplicationException(e);
            }
    });
}
```

The `Request.entity` method accepts an `NioReaderHandler.read(NioInputStream)` handler and, optionally, an `NioCompletionHandler` or `NioErrorHandler`, respectively, as an additional listener. Calling `isFinished()` on the input stream reveals whether more data is available.

By using these approaches, you can realize NIO by specifying asynchronous handlers for the data sent with the request or response.

Client-side NIO. Similar to the reactive integration in the client that was shown earlier, the NIO feature is realized as a separate invoker, callable by `nio()` on the `Invocation.Builder`, which returns an `NioInvoker` that is capable of handling non-blocking requests by `NioWriterHandler` or `NioReaderHandler` callbacks.

The following examples write and read data from the client side in a non-blocking way.

```
final InputStream in = // ...
final byte[] buffer = new byte[1000];

target.request(
    MediaType.APPLICATION_OCTET_STREAM).nio().post(
        out -> {
            try {
                final int length = in.read(buffer);
                if (length >= 0) {
                    out.write(buffer, 0, length);
                    return true;
                }
                in.close();
                return false;
            } catch (IOException e) {
                throw new WebApplicationException(e);
            }
        });
final OutputStream out = // ...
final byte[] buffer = new byte[1000];

target.request().accept(
    MediaType.APPLICATION_OCTET_STREAM).nio().get(
        in -> {
            try {
                if (in.isFinished()) {
                    out.close();
                    // processing the output further...
                } else {
                    final int length = in.read(buffer);
                    out.write(buffer, 0, length);
                }
            } catch (IOException e) {
                throw new WebApplicationException(e);
            }
        });
```

As on the server side, these builders also optionally accept an `NioErrorHandler`.

The JAX-RS implementation for both the client and the server takes care of the internal processing of the NIO by using `java.nio.*` features so that you don't need to worry about scheduling or handling the asynchronicity.

Version 2.1 of JAX-RS will ship with new features in Java EE 8.

Integration with Other Specifications

Besides the previously mentioned features that will affect the JAX-RS programming model, there will also be a few improvements in how the rest of the specifications work together within Java EE.

Because Java EE 8 will be the first version to have native support for JSON binding via JSON-B, it is crucial to be able to use that specification seamlessly with JAX-RS. As of today, binding XML to POJOs can be realized via Java Architecture for XML Binding (JAXB) mapping—most likely with a declarative approach using annotations. It works equally well to read from and write to POJOs that are mapped to JSON with JAX-RS using the JSON-B provider:

```
@Path("{id}")
@GET
public Example getExample(
    @PathParam("id") long id) {
    // ...
}

@POST
public void createExample(Example example) {
    // ...
}
public class Example {

    @JsonbTransient
    private long id;

    @JsonbProperty("hello")
    private String greeting;

    // getters & setters omitted
}
```

Another possible improvement is how other specifications that work with JAX-RS tackle Contexts and Dependency Injection (CDI), or JSR 330. Objects managed by the JAX-RS implementation are injected into the resource classes and methods mainly via `@Context` and field `Params`.

Conclusion

As of today, JAX-RS 2.0 is a very usable and widely adopted specification in Java EE. Version 2.1, which will ship with Java EE 8, will be improved by the small enhancements and new features I've described here. Other changes that are already in the API's snapshot version can be found in the repositories on GitHub. </article>

Sebastian Daschner (@DaschnerS) is a Java EE freelancer based in Munich, Germany. He has more than six years of Java experience, contributes to various open source projects, is a Java Champion, and participates in the JCP as a JSR 370 Expert Group member. Daschner evangelizes computer science practices on his blog. When not working with Java, he loves to travel the world by plane or motorbike.

learn more

JSR 370 web page

MICHAEL KÖLLING

The Evolving Nature of Interfaces

Understanding multiple inheritance in Java

In the "New to Java" series, I try to provide benefit by picking topics that invite a deeper understanding of the conceptual background of a language construct. Often, novice programmers have a working knowledge of a concept—that is, they can use it in many situations, but they lack a deeper understanding of the underlying principles that would lead to writing better code, creating better structures, and making better decisions about when to use a given construct. Java *interfaces* are just such a topic.

In this article, I assume that you have a basic understanding of inheritance. Java interfaces are closely related to inheritance, as are the `extends` and `implements` keywords. So, I will discuss why Java has two different inheritance mechanisms (indicated by these keywords), how abstract classes fit in, and what various tasks interfaces can be used for.

As is so often the case, the story of these features starts with some quite simple and elegant ideas that lead to the definition of concepts in early Java versions, and the story gets more complicated as Java advances to tackle more-intricate, real-world problems. This leads to the introduction of default methods in Java 8, which muddy the waters a bit.

A Little Background on Inheritance

Inheritance is quite straightforward to understand in principle: a class can be specified as an extension of another class. In such a case, the present class is called a *subclass*, and the class it's extending is called the *superclass*. Objects of the subclass have all the properties of both the superclass and the subclass. They have all fields defined in either subclass or superclass and also all methods from both. So far, so good.

Inheritance is, however, the equivalent of the Swiss Army knife in programming: it can be used to achieve some very diverse goals. I can use inheritance to reuse some code I have written before, I can use it for subtyping and dynamic dispatch, I can use it to separate specification from implementation, I can use it to specify a contract between different parts of a system, and I can use it for a variety of other tasks. These are all important, but very different, ideas. It is necessary to understand these differences to get a good feel for inheritance and interfaces.

Type Inheritance Versus Code Inheritance

Two main capabilities that inheritance provides are the ability to inherit code and the ability to inherit a type. It is useful to separate these two ideas conceptually, especially because standard Java inheritance mixes them together. In Java, every class I define also defines a type: as soon as I have a class, I can create variables of that type, for example.

When I create a subclass (using the `extends` keyword), the subclass inherits both the code and the type of the superclass. Inherited methods are available to be called (I'll refer to this as "the code"), and objects of the subclass can be used in places where objects of the superclass are expected (thus, the subclass creates a subtype).

Let's look at an example. If `Student` is a subclass of `Person`, then objects of class `Student` have the type `Student`, but they also have the type `Person`. A student is a person. Both the code and the type are inherited.

PHOTOGRAPH BY
JOHN BLYTHE

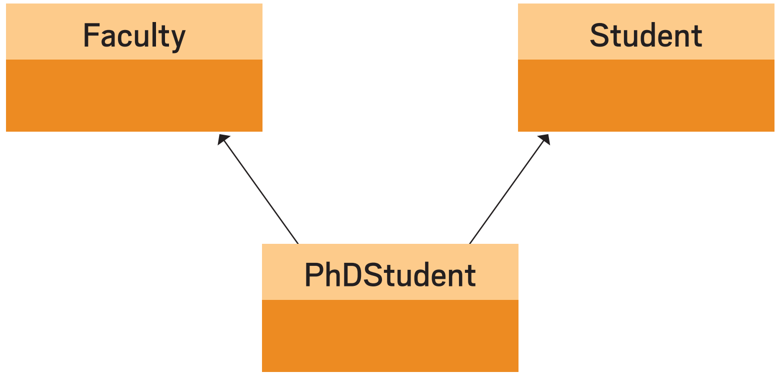


Figure 1. An example of multiple inheritance

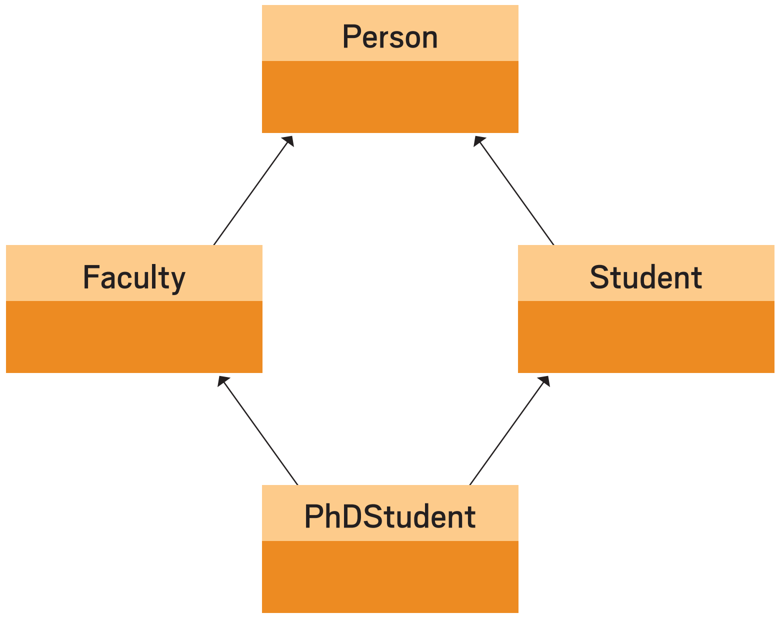


Figure 2. An example of diamond inheritance

The decision to link type inheritance and code inheritance in Java is a language design choice: it was done because it is often useful, but it is not the only way a language can be designed. Other programming languages allow inheriting code without inheriting the type (such as C++ *private inheritance*) or inheriting type without code (which Java also supports, as I explain shortly).

Multiple Inheritance

The next idea entering the mix is *multiple inheritance*: a class may have more than one superclass. Let me give you an example: PhD students at my university also work as instructors. In that sense, they are like faculty (they are instructors for a class, have a room number, a payroll number, and so on). But they are also students: they are enrolled in a course, have a student ID number, and so on. I can model this as multiple inheritance (see **Figure 1**).

`PhDStudent` is a subclass of both `Faculty` and `Student`. This way, a PhD student will have the attributes of both students and faculty. Conceptually this is straightforward. In practice, however, the language becomes more complicated if it allows multiple inheritance, because that introduces new problems: What if both superclasses have fields with the same name? What if they have methods with the same signature but different implementations? For these cases, I need language constructs that specify some solution to the problem of ambiguity and name overloading. However, it gets worse.

Diamond Inheritance

A more complicated scenario is known as *diamond inheritance* (see **Figure 2**). This is where a class (`PhDStudent`) has two superclasses (`Faculty` and `Student`), which in turn have a common superclass (`Person`). The inheritance graph forms a diamond shape.

Now, consider this question: if there is a field in the top-level superclass (`Person`, in this case), should the class at the bottom (`PhDStudent`) have one copy of this field or two? It inherits this field twice, after all, once via each of its inheritance branches.

The answer is: it depends. If the field in question is, say, an ID number, maybe a PhD student should have two: a student ID and a faculty/payroll ID that might be a different number. If the field is, however, the person's family name, then you want only one (the PhD student has only one family name, even though it is inherited from both superclasses).

In short, things can become very messy. Languages that allow full, multiple inheritance need to have rules and constructs to deal with all these situations, and these rules are complicated.

Type Inheritance to the Rescue

When you think about these problems carefully, you realize that all the problems with multiple inheritance are related to inheriting code: method implementations and fields. Multiple code inheritance is messy, but multiple type inheritance causes no problems. This fact is coupled with another observation: multiple code inheritance is not terribly important, because you can use *delegation* (using a reference to another object) instead, but multiple subtyping is often very useful and not easily replaced in an elegant way.

That is why the Java designers arrived at a pragmatic solution: allow only single inheritance for code, but allow multiple inheritance for types.

Interfaces

To make it possible to have different rules for types and code, Java needs to be able to specify types without specifying code. That is what a Java interface does.

Interfaces specify a Java type (the type name and the signatures of its methods) without specifying any implementation. No fields and no method bodies are specified. Interfaces can contain constants. You can leave out the modifiers (`public static final` for constants and `public` for methods) —they are implicitly assumed.

This arrangement provides me with two types of inheritance in Java: I can inherit a class (using `extends`), in which I inherit both the type and the code, or I can inherit a type only (using `implements`) by inheriting from an interface. And I can now have different rules concerning multiple inheritance: Java permits multiple inheritance for types (interfaces) but only single inheritance for classes (which contain code).

Benefits of Multiple Inheritance for Types

The benefits of allowing the inheritance of multiple types—essentially of being able to declare that one object can be viewed as having a different type at different times—is quite easy to see. Suppose you are writing a traffic simulation, and in it you have objects of class `Car`. Apart from cars, there are other kinds of active objects in your simulation, such as pedestrians, trucks, traffic lights, and so on. You may then have a central collection in your program—say, a `List`—that holds all the actors:

```
private List<Actor> actors;
```

`Actor`, in this case, could be an interface with an `act` method:

```
public interface Actor
{
    void act();
}
```

Your `Car` class can then implement this interface:

```
class Car implements Actor
{
    public void act()
    {
        ...
    }
}
```

Note that, because `Car` inherits only the type, including the signature of the `act` method, but no code, it must itself supply the code to implement the type (the implementation of the `act` method) before you can create objects from it.

So far, this is just single inheritance and could have been achieved by inheriting a class. But imagine now that there is also a list of all objects to be drawn on screen (which is not the same as the list of actors, because some actors are not drawn, and some drawn objects are not actors):

```
private List<Drawable> drawables;
```

You might also want to save a simulation to permanent storage at some point, and the objects to be saved might, again, be a different list. To be saved, they need to be of type `Serializable`:

```
private List<Serializable> objectsToSave;
```

In this case, if the `Car` objects are part of all three lists (they act, they are drawn, and they should be saved), the class `Car`

can be defined to implement all three interfaces:

```
class Car implements Actor, Drawable, Serializable ...
```

Situations like this are common, and allowing multiple supertypes enables you to view a single object (the car, in this case) from different perspectives, focusing on different aspects to group them with other similar objects or to treat them according to a certain subset of their possible behaviors.

Java's GUI event-processing model is built around the same idea: event handling is achieved via event listeners—interfaces (such as `ActionListener`) that often just implement a single method—so that objects that implement it can be viewed as being of a listener type when necessary.

Abstract Classes

I should say a few words about *abstract classes*, because it is common to wonder how they relate to interfaces. Abstract classes sit halfway between classes and interfaces: they define a type and can contain code (as classes do), but they can also have *abstract methods*—methods that are specified only, but not implemented. You can think of them as partially implemented classes with some gaps in them (code that is missing and needs to be filled in by subclasses).

In my example above, the `Actor` interface could be an abstract class instead. The `act` method itself might be abstract (because it is different in each specific actor and there is no reasonable default), but maybe it contains some other code that is common to all actors.

In this case, I can write `Actor` as an abstract class, and the inheritance declaration of my `Car` class would look like this:

```
class Car extends Actor implements Drawable, Serializable
...
```

If I want several of my interfaces to contain code, turning them all into abstract classes does not work. As I stated before, Java allows only single inheritance for classes (that means only one class can be listed after the `extends` keyword). Multiple inheritance is for interfaces only.

There is a way out, though: *default methods*, which were introduced in Java 8. I'll get to them shortly.

Empty Interfaces

Sometimes you come across interfaces that are empty—they define only the interface name and no methods. `Serializable`, mentioned previously, is such an interface. `Cloneable` is another. These interfaces are known as *marker interfaces*. They mark certain classes as possessing a specific property, and their purpose is more closely related to providing metadata than to implementing a type or defining a contract between parts of a program. Java, since version 5, has had annotations, which are a better way of providing metadata. There is little reason today to use marker interfaces in Java. If you are tempted, look instead at using annotations.

A New Dawn with Java 8

So far, I have purposely ignored some new features that were introduced with Java 8. This is because Java 8 adds functionality that contradicts some of the earlier design decisions of the language (such as "only single inheritance for code"), which makes explaining the relationship of some constructs quite difficult. Arguing the difference between and justification for the existence of interfaces and abstract classes, for instance, becomes quite tricky. As I will show in a moment, interfaces in Java 8 have been extended so that they become more similar to abstract classes, but with some subtle differences.

In my explanation of the issues, I have taken you down the historical path—explaining the pre-Java 8 situation first

and now adding the newer Java 8 features. I did this on purpose, because understanding the justification for the combination of features as they are today is possible only in light of this history.

If the Java team were to design Java from scratch now, and if breaking backward compatibility were not a problem, they would not design it in the same way. The Java language is, however, not foremost a theoretical exercise, but a system for practical use. And in the real world, you must find ways to evolve and extend your language without breaking everything that has been done before. Default methods and static methods in interfaces are two mechanisms that made progress possible in Java 8.

Evolving Interfaces

One problem in developing Java 8 was how to evolve interfaces. Java 8 added lambdas and several other features to the Java language that made it desirable to adapt some of the existing interfaces in the Java library. But how do you evolve an interface without breaking all the existing code that uses this interface?

Imagine you have an interface MagicWand in your existing library:

```
public interface MagicWand
{
    void doMagic();
}
```

This interface has already been used and implemented by many classes in many projects. But you now come up with some really great new functionality, and you would like to add a really useful new method:

```
public interface MagicWand
{
    void doMagic();
    void doAdvancedMagic();
}
```

If you do that, then all classes that previously implemented this interface break, because they are required to provide an implementation for this new method. So, at first glance, it seems you are stuck: either you break existing user code (which you don't want to do) or you're doomed to stick with your old libraries without a chance to improve them easily. (In reality, there are some other approaches that you could try, such as extending interfaces in subinterfaces, but these have their own problems, which I do not discuss here.) Java 8 came up with a clever trick to get the best of both worlds: the ability to add to existing interfaces without breaking existing code. This is done using *default methods* and *static methods*, which I discuss next.

Default Methods

Default methods are methods in interfaces that have a method body—the default implementation. They are defined by using the default modifier at the beginning of the method signature, and they have a full method body:

```
public interface MagicWand
{
    void doMagic();
    default void doAdvancedMagic()
    {
        ... // some code here
    }
}
```

Classes that implement this interface now have the chance to provide their own implementation for this method (by overriding it), or they can completely ignore this method,

in which case they receive the default implementation from the interface. Old code continues to work, while new code can use this new functionality.

There is little reason today to use marker interfaces in Java. If you are tempted, look instead at using annotations.

Static Methods

Interfaces can now also contain static methods with implementations. These are defined by using the usual `static` modifier at the beginning of the method signature. As always, when writing interfaces, the `public` modifier may be left out, because all methods and all constants in interfaces are always public.

So, What About the Diamond Problem?

As you can see, abstract classes and interfaces have become quite similar now. Both can contain abstract methods and methods with implementations, although the syntax is different. There are still some differences (for instance, abstract classes can have instance fields, whereas interfaces cannot), but these still leave the central point: since the release of Java 8, you have multiple inheritance (via interfaces) that can contain code!

At the beginning of this article I pointed out how the Java designers treaded very carefully to avoid multiple code inheritance because of possible problems, mostly related to inheriting multiple times and to name clashes. So what is the situation now?

As usual, the Java designers have settled on the following pragmatic rules to deal with these problems:

- Inheriting multiple abstract methods with the same name is not a problem—they are viewed as the same method.
- Diamond inheritance of fields—one of the difficult problems—is avoided, because interfaces still are not allowed to contain fields that are not constants.
- Inheriting static methods and constants (which are also static by definition) is not a problem, because they are prefixed by the interface name when they are used, so their names do not clash.
- Inheriting from different interfaces multiple default methods with the same signature and different implementations is a problem. But here Java chooses a much more pragmatic solution than some other languages: instead of defining a new language construct to deal with this, the compiler just reports an error. In other words, it's your problem. Java just tells you, "Don't do this."

Conclusion

Interfaces are a powerful feature in Java. They are useful in many situations, including for defining contracts between different parts of the program, defining types for dynamic dispatch, separating the definition of a type from its implementation, and allowing for multiple inheritance in Java. They are very often useful in your code; you should make sure you understand their behavior well.

The new interface features in Java 8, such as default methods, are most useful when you write libraries; they are less likely to be used in application code. However, the Java libraries now make extensive use of them, so make sure you know what they do. Careful use of interfaces can significantly improve the quality of your code. </article>

Michael Kölling is a Java Champion and a professor at the University of Kent, England. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and he is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is also a Distinguished Educator of the ACM.

BRIAN FRANK

Fantom Programming Language

A language that runs on the JVM and JavaScript VMs and delivers excellent UI-building capabilities

Fantom is a programming language derived from mainstream languages such as Java and C#. The primary design goal from its start in 2005 was portability between heterogeneous runtime environments, specifically the JVM and browser JavaScript VMs. The Fantom compiler and extensive standard library ensure that code written in Fantom works exactly the same on a Java server and in a browser client. Although many JVM languages now have efforts around providing a JavaScript port, few were designed from the ground up to solve this challenging problem. You will find the goal for seamless portability woven into every aspect of the Fantom platform.

Fantom also supports a novel approach to immutability and concurrency. The type system and runtime work in concert to provide guaranteed immutability. An actor model for concurrency leverages the type system to ensure that mutable state is never shared between threads.

Fantom is designed to be a single language for writing both server-side and browser-client-side code. On the client side, it has a new library called domkit for building rich HTML5 user experiences.

This article provides an in-depth look into several of these prominent features: portability; immutability; actor concurrency; and Fantom's HTML5 toolkit, domkit.

Portability

True portability requires both compiler support and library support. If your language is portable but all the libraries you use are not, then you haven't really solved the problem. For this reason, Fantom has its own standard library, which is equally important as the language itself.

Fantom was conceived to provide both compiler support and a standard library to port between three different runtime environments: Java, .NET, and JavaScript VMs. Although we prototyped a Fantom runtime for .NET, it isn't currently supported. Today, all development is done only for the Java and JavaScript runtimes. These two runtimes have been used in production for commercial software for more than six years—making them very mature and robust.

The Fantom compiler supports a pluggable architecture. A single pass of the compiler is used to simultaneously generate bytecode for the Java VM and transpiled JavaScript source for browser environments. Once a Fantom module is built, a single module file can be deployed with everything required to use the code on a Java server and in a browser.

The compiler also provides a mechanism for classes and methods to be implemented "natively." Native code is implemented in Java and/or JavaScript to create the low-level APIs that access functionality from the underlying VM. These low-level APIs include core types (Str, Int, DateTime), collections, I/O, networking, concurrency, and Document Object Model (DOM) access. With these APIs in place, most Fantom modules are written in 100 percent Fantom and have guaranteed portability between Java and JavaScript.

Immutability

One of Fantom's guiding principles has been to have a type

system and standard library designed to support immutability. Although immutable data structures are common in functional languages, they typically aren't built into mainstream object-oriented languages.

Fields in Fantom may be marked `const` to guarantee that they are immutable:

```
const Point pt
```

This snippet of code declares a field of type `Point` that has a compile-time guarantee of *deep immutability*. Deep immutability means that we not only guarantee that the `pt` reference is set only once, but we also guarantee that everything the `Point` instance references is also immutable. Contrast this with a Java `final` field that provides only *shallow immutability*: the reference cannot be changed, but there is no guarantee of immutability for the referenced objects.

Deep immutability is guaranteed by the type system using `const` classes:

```
const class Point
{
  new make(Int x, Int y) { this.x = x; this.y = y }
  const Int x
  const Int y
}
```

In the preceding example, the entire class is marked `const`, which uses compile-time checking to ensure that instances of the class are deeply immutable. Specifically, this means that all fields are marked `const` and set only in the constructor (the method named `make` annotated with the `new` keyword).

The standard library provides many conveniences for using a mutable data structure to build up a collection and then freezing it as an immutable instance:

```
list := [,]         // create an empty list
list.add("a")       // add some items
list.toImmutable    // returns immutable copy of list
```

The snippet above uses an immutable list to build up a result, and then uses the `toImmutable` method to efficiently get an immutable copy of the list. Similar APIs are available for other collection types, in-memory byte buffers, and even functions. Once we have immutable instances, we can safely assign them to `const` fields and share them between threads.

Concurrency

Fantom's approach to concurrency is built around this key concept: make it impossible to share mutable state between threads. Enforcing this restriction makes it much easier to reason about concurrency, avoids deadlocks and race conditions, and greatly increases robustness. Fantom achieves this goal with a built-in concurrency model based on *actors*. There is no synchronized or volatile mechanism in Fantom; there are only actors.

Actors are lightweight objects designed to asynchronously process work on a thread pool. Interaction with actors is done via a message queue. Client code sends a message to an actor and is returned a Future, which the client can optionally use to block until the result is ready. Messages sent to actors are queued. Once the actor framework detects that an actor has pending messages, the actor is scheduled to a thread to process its message queue via the `receive` method. Actors are guaranteed to receive their messages in order and to safely execute within one thread.

Let's look at a simple example for an actor that receives integer messages and returns the mathematical square. Here is our actor class:

```
const class SquareActor : Actor
{
```

```
  new make(ActorPool pool) : super(pool) {}

  override Obj? receive(Obj? msg)
  {
    i := msg as Int ?:
            throw ArgErr("Expecting Int msg")
    return i * i
  }
}
```

There are several things to note in the preceding code. Actors are required to be `const` classes (immutable). The constructor takes an `ActorPool`, which manages the threading for this actor. The `receive` method is overridden to process incoming messages. Notice that `receive` takes and returns an `Obj?` type. This syntax illustrates a *nullable type*, which means the type system allows null to be used for these signatures. Conversely, the type `Obj` (without a question mark) is non-nullable and enforced by the type system to never be null. The implementation of `receive` casts the object to an integer and returns the square. Note the use of the `as` keyword, which works as it does in C#, and the elvis operator (`?:`), which is syntax sugar for this code:

```
  Int i = msg instanceof Int ? (Int)msg : null;
  if (i == null) throw new ArgErr("Expecting Int msg");
```

Now let's look at how we can use this actor:

```
pool   := ActorPool()
actor  := SquareActor(pool)
future := actor.send(3)
val    := future.get
```

Let's digest the code above. First, I create an `ActorPool` instance, which manages the thread pool. Next, I create an instance of `SquareActor`, which I bind to the pool. The third line sends a message to the actor. What happens under the covers is that the actor queues the messages, allocates a thread from the pool, and processes the message on a background thread using the `receive` method. Meanwhile, the client code has blocked on the `Future.get` method. When the result becomes available, then the `val` variable will be assigned the result of 9.

The Fantom actor framework enforces that all messages are immutable using the `const` class type system. This approach ensures that data mutations are restricted to a single actor thread and avoids the need for synchronization. Actors provide an elegant, robust alternative to the mainstream concurrency techniques used in languages such as Java.

domkit

One of Fantom's most recent developments is a new HTML5 widget toolkit named domkit. It provides a rich library to build highly polished HTML5 UIs with a design familiar to anyone who has experience with a traditional toolkit such as Swing.

Fantom provides two levels of abstraction for working in HTML5. The dom module provides a statically typed, low-level API to access and manipulate the DOM, user input events, CSS, and XHR (XML HTTP Request). Under the covers, it provides the JavaScript glue to the HTML5 platform.

The domkit module provides a much higher level of abstraction. It is built entirely in Fantom on top of the dom module. It provides DOM-backed widgets using a familiar widget design including Menus, Tables, Trees, Buttons, and Dialogs. domkit leverages CSS for styling and layout, but allows you to work at a higher level of abstraction.

Let's look at an example for a table widget. In Swing, if you wanted to show information as a table you would create an instance of `TableModel` and render it using a `JTable` instance. Browsers don't come with anything like `JTable`, but domkit comes with a `Table` and `TableModel` API, which work

Col 0	Col 1	Col 2	Col 3	Col 4
0 x 0	0 x 1	0 x 2	0 x 3	0 x 4
1 x 0	1 x 1	1 x 2	1 x 3	1 x 4
2 x 0	2 x 1	2 x 2	2 x 3	2 x 4

Figure 1. A sample table created with Fantom domkit

just like Swing. The domkit `Table` automatically handles all the thorny issues: efficiently mapping the model to DOM elements (but only for the visible rows), scrolling, column sorting, single/multiple selection, and many other features.

Let's look at some real code to illustrate how to build a table using domkit:

```
@Js
class MyTableModel : TableModel
{
  override Int numCols() { 5 }
  override Int numRows() { 3 }
  override Void onHeader(Elem e, Int c) {
     e.text = "Col $c" }
  override Void onCell(
     Elem e, Int c, Int r, TableFlags f)
        { e.text = "$r x $c" }
}
```

This creates a subclass of the domkit `TableModel`. It defines the number of rows and columns for the model and provides a callback for how to render the column headers and cells. Now let's see how I put it all together to create a table:

```
table := Table
{
  model = MyTableModel()
  sel.multi = true
  onAction |t| {
      echo("onAction: $t.sel.indexes") }
  onSelect |t| {
      echo("onSelect: $t.sel.indexes") }
}
```

This code creates an instance of `Table` using the `model` class and sets multiple selection to be enabled. Then, I add some event handlers for action (double click) and selection changes that echo to `stdout` the selected row indexes. **Figure 1** shows what the table looks like in a browser.

If you have experience building HTML5 UIs and miss the higher-level abstractions that a widget toolkit such as Swing provides, then domkit might be just the technology for you.

Conclusion

In this article, I examined four key features of Fantom: portability, immutability, actor concurrency, and domkit. This only briefly touches on the Fantom language, libraries, and tools. If you would like to learn more, visit our website, where you can find documentation, a community forum, an active mailing list, and links to downloads as well as our BitBucket repo. </article>

Brian Frank is the founder and president of SkyFoundry, a software company specializing in IoT data collection, analysis, and visualization. Brian and his brother, Andy Frank, have been developing the Fantom platform since 2005. Brian also serves as the technical lead for project-haystack.org, an open source project for defining data models and formats in the IoT space.

learn more

Why Fantom?

SIMON ROBERTS

Quiz Yourself

More subtle questions from an author of the Java certification tests

I've put together more problems that simulate questions from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise. [Readers wishing basic instruction should consult the "New to Java" column, which appears in every issue. —*Ed.*] These questions might require careful deduction to obtain the right answer.

Question 1. Given this code:

```
public IntSupplier doStuff(int [] vals, int i) {
  // line n1
  return () -> vals[i];
}
```

Which two are true? Choose two.

- **a.** The code compiles.
- **b.** The argument list must be changed to `(final int[] vals, final int i)` to allow the code to compile.
- **c.** If the code `if (vals[0] < 0) vals[i] = 0;` is added at line n1, the argument list must be changed to `(final int[] vals, final int i)` to allow the code to compile.
- **d.** The code `if (vals[0] < 0) vals[i] = 0;` can be added at line n1 without causing compilation errors.
- **e.** The code `vals = Arrays.copyOf(vals, vals.length);` can be added at line n1 without causing compilation errors.

Question 2. Given the following code:

```
public class Wrapper {
  public class Wrapped {}
}
```

Which is true? Choose one.

- **a.** An instance of `Wrapped` can be created only by code inside the class `Wrapper`.
- **b.** An instance of `Wrapped` can be created using the expression `new Wrapper.Wrapped()`.
- **c.** An instance of `Wrapped` can be created using the expression `new Wrapper().Wrapped()`.
- **d.** An instance of `Wrapped` can be created using the expression `new Wrapper().new Wrapped()`.
- **e.** An instance of `Wrapped` can be created using the expression `new Wrapper::Wrapped()`.

Question 3. You wish to calculate the sum of the numbers in a stream and also print out each one. So far, you have this code:

```
public static int sumAndPrint(IntStream is) {
  int total = 0;
  is.parallel()
    .peek(v -> total += v)
    .forEach(System.out::println);
  return total;
}
```

Which is true? Choose one.

a. The code reliably returns the correct sum of the numbers as it is.

b. The code would reliably return the correct sum of the numbers if the body of the method were changed to this:

```
int[] total = {0};
is.peek(v -> total[0] += v)
  .forEach(System.out::println);
return total[0];
```

c. The code would reliably return the correct sum of the numbers if the type of `total` were changed to `Integer`.

d. The code would reliably return the correct sum of the numbers if the call to `.parallel()` were moved after the call to `.peek(v -> total += v)`.

e. The code would reliably return the correct sum of the numbers if the type of `total` were changed to `LongAdder` and if the lambda in `peek` were changed accordingly.

Question 4. Given this code:

```
public static void main(String[] args) {
  ExecutorService es = Executors.newFixedThreadPool(2);
  Callable<String> job = () -> {
    Thread.sleep(5000); // line n1
    return "Hello";
  };
  es.submit(job);
  es.submit(job);
  Future<String> handle = es.submit(job); // line n2
  System.out.println("submitted");
  String message = handle.get(); // line n3
  System.out.println("> " + message);
  System.exit(0);
}
```

Which is true? Choose one.

a. Compilation fails due to an error at line n1.

b. Compilation fails due to an error at line n3.

c. The program throws a `RejectedExecutionException` because there are no available threads at line n2.

d. The program immediately prints `submitted`, and then after a 10-second pause, it prints `> Hello`.

e. The program pauses five seconds, then prints `submitted`, and then after a further five-second pause, it prints `> Hello`.

Question 5. Given that the current working directory of the following program contains this tree:

```
.
├── one/
│   ├── a/
│   │   └── t1.txt
│   └── b/
└── two/
    ├── c/
    └── t2.txt
```

where `one`, `two`, `a`, `b`, and `c` are directories, and the program code is this:

```
public static void main(String[] args) throws Throwable {
  Files
    .find(Paths.get("."), 1, (p,a)->a.isDirectory())
    .forEach(System.out::println);
}
```

What is the output? Choose one.

a. `-`

b.
```
.
./one
./two
```

c.
```
./one
./one/b
./one/a
./two
./two/c
```
d.
```
./one
./one/b
./one/a
./two
./two/c
./two/t2.txt
```
e.
```
./two/t2.txt
```

Question 1. The correct answers are options A and D. The first decision is probably to determine whether the code compiles or whether the change outlined in option B is necessary. This one is interesting, because it depends critically on the version of Java in use.

When a method-local variable is referenced from a class or lambda expression enclosed in the method's scope, a potential problem arises. A method-local variable ceases to exist when the method returns. However, the lifetime of the object created by a nested class or lambda is potentially much longer. Java's particular solution to this situation (sometimes called a *closure* or a *captured variable*) is to insist that the method-local variable must never be changed. Given that restriction, it's possible to simply take a copy of the variable and embed it in the longer-lived object. If the variable can't change, a copy is as good as the original.

From Java 1.1—which introduced nested classes—through Java 7, any method-local variable accessed from a nested class had to be labeled `final`. This means that option B would be correct for Java 7. However, Java 8 changed the requirements a little, and the keyword `final` is no longer required (although the variable must be treated as though `final` were present).

It turns out that the compiler has had the ability since the beginning to determine if the rules of `final` are properly adhered to; that's been important because the compiler must issue an error if you try to change a `final` variable. Also, a `final` variable can permit some optimizations in the generated code, and the compiler has been doing neat tricks using this for some time. Of course, those optimizations can be done based on a variable *behaving like* a `final`, not on whether the programmer labeled the variable as such. The compiler has a notion of "effectively `final`"—the programmer didn't say it's `final`, but it is, so the compiler treats it as such.

As mentioned, Java 8 changed the rules for those method-local variables. Today, they must be effectively `final`, but they do not have to be marked as such. Consequently, the code does in fact compile in a Java 8 environment (and this article is written about the Java 8 certification exam). So, option A is correct and option B is not.

Option C builds on this question of finality. What happens if the code `if (vals[0] < 0) vals[i] = 0;` is added at line n1? Well, `vals` is a pointer to an array, and the pointer is *not* changed by this, so `vals` remains effectively `final`. This is an important point about final-ness for variables, and indeed, one of the key reasons that Java 1.1 added the `final` restriction. A `final` variable that is of reference type cannot be changed to point at a different object, but the contents of the object (or array, in this case) are not protected by the `final` keyword, so not much of a restriction is being imposed. Immutable objects such as `Integer` and `String` are special, not because `final` makes any special considerations, but because

they are intrinsically immutable. Anyway, in this case, `vals` is still effectively `final`; only the contents of the array are changed, and adding the code is successful without any other changes needed. Hence, option C is false.

In fact, the preceding argument shows that option D is also true. The code can be added, and no changes are needed.

By the same token, however, option E is false. Assigning a new value to `vals`—that is, pointing `vals` at a different array (although it's a copy of the original, which might seem like a good thing in general)—is sufficient to break the "effectively `final`" requirement, and this code will not compile.

Question 2. The correct answer is option D. This is a question with less depth and complexity than many. The terminology sometimes can be confusing, but to be frank, that terminology isn't really very important. Java uses the term *nested class* to describe classes defined inside other classes. So, `Wrapped` is a nested class. However, such a class can be labeled `static`. If it's static then, just like any other static element, it's associated with the class as a whole rather than with any specific instance. On the other hand, a nested class that is not labeled `static` is properly called an *inner class* (though, as I suggested, the terminology is commonly used very loosely, and no exam question would test such a distinction of terminology). What matters, however, is that a nonstatic nested class—that is, an inner class in the official sense—actually has direct access, through a hidden reference that acts similarly to `this`, to the fields of a particular instance of the enclosing class. It's as if the inner class is part of the enclosing instance.

Of course, that means that when the instance of the inner class is created, there must be an enclosing instance for it to belong to. This is why option D turns out to be the required syntax. The general approach is that there must be some kind of prefix for the `new` operation. The prefix can be explicit as it is here; first, you build the instance of `Wrapper`, and then you invoke `new` upon that instance to create the `Wrapped` object within.

The operation could be done in two stages. If `myWrapper` were a variable of type `Wrapper`, you could initialize `myWrapper`, and then use `myWrapper.new Wrapped()`. However, in this case, it was done all at once.

Option A isn't true; you can create instances outside of `Wrapper`. That said, it's probably much more common, and often a better design, to create the inner classes inside the enclosing class using something along the lines of a factory method. It's even likely that the inner class might be private and known to the outer world only by some public interface that it implements. While acknowledging that there are, of course, special cases, the general notion of encapsulation suggests that fiddling with the inner class should mostly be done by the outer one. The bottom line for this question, though, is that option A is incorrect.

All the other syntax offerings are incorrect for the purpose at hand. Option B would work if `Wrapped` were a nested class, but not if it were an inner class. That is, if `Wrapped` were a static class, then option B would be correct. But, because the conditions are not so, it's incorrect.

Option C is not a valid construction of anything. It could be valid as an invocation of a method called `Wrapped()` that's a member of the `Wrapper` class. Such a method contradicts normal Java style conventions, which reserve capital first letters for classes and interfaces and call for lowercase letters for methods. However, it's actually possible for that method to coexist with the class of the same name, creating a situation where option C could compile and return a new object of `Wrapped` type. However, that's not going to work unless the method is added to the class, and because the entire class is shown in the example, you can rule out this line of reasoning, which is based on some crazy inappropriate broken style conventions.

Option E is just made-up syntax. It's not a method reference, because it includes parentheses. It doesn't have any context for a lambda's type to be determined to support the

use of method references anyway. Further, the `new` keyword should follow the double colon in a method reference. Option E is just wrong.

Question 3. The correct answer is option E. This question investigates the concurrent mode of stream processing and the rules that ensure it runs correctly and in a thread-safe fashion. A common recommendation is that stream processing, if there's any chance of the stream being executed in concurrent mode, should not mutate any shared data. Such an approach is a great way to ensure that concurrency problems do not arise. However, the actual rule is somewhat less restrictive. If shared state will be modified, then the programmer must ensure that those modifications are safe. In this case, using a `LongAdder` would achieve this goal, and option E is the correct answer. Interestingly, this situation could also be served by using the `AtomicInteger`, but the `LongAdder` was deliberately created to support many concurrent mutations in a scenario where relatively few reads are performed. Both would be functionally correct, but the adder will be more scalable.

Let's look at the wrong answers and see what might be interesting about those. First, why doesn't this code compile? The variable `total` is a method-local variable, and for that to be used in an enclosed lambda, it must be `final` or effectively `final`. Consequently, because it is not `final`, and it cannot be because it is mutated, the code as it stands will not compile. This is why option A is incorrect.

Option B looks tempting, and in many cases it would probably work. First, it uses an array of one `int` to accumulate the total. This successfully sidesteps the problem of a `final` variable, because the variable is a pointer to the array and is, indeed, now effectively `final`. However, even though the call to `parallel` was also removed from the code, the stream was received as an argument to the method, so it's not safe to assume that the stream is now running sequentially. It could have been set to parallel by the caller. In that situation, the code remains unsafe from a concurrency perspective. So, option B is incorrect, because the behavior would still not be reliable.

Option C is unworkable; `Integer` objects are immutable, and given that `total` must be effectively `final`, there's really no way that such a suggestion could result in correct counting.

Option D might seem tempting. Presumably the idea is that if the mutation operation, performed by `peek`, can be performed sequentially, there will be no concurrency issue with respect to the updates to `total`. Unfortunately, this fails for two reasons: first and most compellingly, the option still doesn't compile because `total` is (still) not effectively `final`. Second, and also important, is the fact that streams don't shift between parallel and serial modes along their length. The whole stream, from end to end, is either parallel or sequential. Therefore, moving the call to `parallel` later in the chain changes absolutely nothing. For both these reasons, option D is also incorrect.

Question 4. The correct answer is option B. There's quite a lot of code in this question. That is unusual in the actual certification exam, but it does happen. This question tests one specific piece of understanding, but it introduces a number of interesting side issues for discussion as distractions.

The code does not compile, simply because the `get` method on a `Future` object throws checked exceptions. In fact, the documentation declares three exceptions, two of which are checked. The unchecked exception is `CancellationException`, which indicates that the job was canceled. Therefore, trying to get its result is meaningless. The checked exceptions are `InterruptedException` and `ExecutionException`. An `InterruptedException` results if the `get` method, which will block if the job hasn't finished yet, is interrupted while waiting. An `ExecutionException` arises if the job itself throws an exception; the cause of the `ExecutionException` is the actual exception thrown by the job.

What about the other options? Why are they wrong? Option A suggests that compilation fails at line n1. That's tempting, because it might appear that the `InterruptedException`, which is thrown by the `Thread.sleep` method, is unhandled. However, the definition of `Callable`, which is the interface that the lambda expression will be implementing, specifies a method `call()`, which is declared as `throws Exception`. So, in fact, this answer doesn't cause a problem, and option A is incorrect.

Option C tries to get you to believe that if an `ExecutorService` is created with a fixed thread count of two, you cannot submit more than two jobs. That's not the case, of course. You can submit more jobs, which will be placed in a queue and executed as one of the threads becomes available. Therefore, option C is also incorrect.

Option D, in which the code prints `submitted` immediately and then pauses 10 seconds before printing `> Hello`, is actually what would happen if the problem of the checked exception in the `get` method at line n3 is fixed. The `submit` calls for all three jobs to complete without noticeable delay. Two jobs start executing promptly, and the third is queued. The first two take 5 seconds to complete, but because they're running on two concurrent threads, that's a total elapsed time of only 5 seconds, not 10. Then, when either of them finishes, the third job starts executing, delays a further 5 seconds, and finally completes, and the `get` call at line n1 receives the message from it.

The timing suggested by option E is testing to see if you know that the `ExecutorService` has a job queue. If it did not, the third job could not complete submitting until a thread was available for it (incidentally, such service configurations are possible). After the pause, the message `submitted` would be printed, and five seconds later, the final output would be shown. However, because a job queue is created when the service is built using `Executors.newFixedThreadPool(...)`, option E is incorrect.

Question 5. The correct answer is option B. This, unfortunately, is one of those questions that depends largely on rote learning, which here has one small redeeming aspect: Those who have spent time playing with an API are likely to use it more fluently. They know what is available, and they spend less time looking it up.

This question is here mainly because it's an interesting excuse to look at a fairly neat, and commonly overlooked, feature of Java's core APIs. The class `java.nio.file.Files` is a utility class filled with static methods that do handy file I/O operations in easy-to-use packages. If you haven't seen it before—it was introduced with Java 7—it's worth spending a few minutes looking over what it offers. The class also gained quite a few additional methods with Java 8, many of which are built to make good use of the Stream API. The `find` method (along with a close cousin, `walk`) is one of those.

The `find` method is clearly modeled on the UNIX `find` utility. It takes a starting directory as its first argument, and begins a recursive search down the file system hierarchy from that starting point. `Path` elements that are found might be passed into the `Stream<Path>` that is the return value of the method.

The next two arguments are the depth of the search and a `BiPredicate` that filters. Let's look at these individually. There's actually a fourth, variable length, argument too, but I'll let you investigate that yourself.

The depth parameter, which is the second argument, specifies the number of directories that the operation will descend into. The starting point is considered to be a directory. Therefore, if a depth of zero is given, only the initial directory will be reported. Specifically, with a depth of zero, no directories, not even the initial one, will be entered. So, if the depth had been zero, the output could only have been as shown in option A. However, a depth of 1 is given, which means that the operation will enter the initial directory and look at what's in there, but it will not descend any further.

This means that the only elements that will even be considered are the two directories one and two, and the file t2.txt.

The next argument is a `BiPredicate<Path,BasicFileAttributes>`. This is called for every path element that is seen to determine whether to include that element in the output. It's a predicate, so it's probably no surprise that if this returns `false`, the item will be removed from the output stream.

Nothing in the question tells you that the arguments are a `Path` and a `BasicFileAttributes`, but because no options include "does not compile," it's a safe assumption that the second lambda argument—`a`, in this case—actually does have a method `isDirectory()`. From that, you can deduce that only directories will be passed into the output, and now the situation goes from three path elements being seen (one, two, and `t2.txt`) to only the two directories moving to the output. Now you know that the correct answer is option B.

It's interesting that this `find` method returns a stream. It's lazy like all streams are supposed to be (though presumably it has a decent-size buffer at the file system interface!).

This actually raises another observation that the question ignores. Stream objects implement `AutoCloseable`, and while in-memory streams don't really need to be closed, those that are attached to operating system resources other than memory (for example, files, network, and database connections) *should* be closed. This can be a subtle but important issue if you're writing a method that receives a stream as an argument: if you don't know how the stream was opened, can you assume that it doesn't need to be closed? In such situations, perhaps you should give the stream the protection of the `try`-with-resources construct. </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies.

THE DANISH JUG

The only Danish Java user group (JUG), Javagruppen, was founded in December 1996. The first official meeting was held in January 1997 and was initiated with these words: "Never before in computer history has a new technology been gaining support the way Java has. Maybe it is because there is a real need for an alternative to C++ and Visual Basic."

During its first years, the JUG functioned as a special interest group for companies working with Java in Denmark, and in 2004 it was reformed as a new legal entity with an elected board and yearly general assemblies. Members of the JUG pay a yearly fee. Until this year, only companies could become members and that membership covered individuals or all employees. Recently, it was decided that private individuals could become members for a reduced fee and students could join for free.

Today the group consists of about 80 paying members, split evenly between individuals and company memberships. The JUG organizes 6 to 10 meetings every year divided between Copenhagen and Aarhus.

The JUG hosts an annual Java developer conference, JDK IO. This year's conference takes place in Copenhagen in the middle of September at the Royal National Library.

Javagruppen has for many years organized programming for kids and has joined the Devoxx4Kids initiative.

Find out more about the JUG by following it on Facebook and Twitter.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese